User Manual

16

Data Flash Access Library

Type T02 (Tiny), European Release

16 Bit Single-chip Microcontroller
RL78 Family

Installer:
RENESAS_FDL_RL78_T02E_Vx.xx

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Technology Corp. website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

R01US0061ED0120
15.04.2015

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics.

8. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti- crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.

9. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
10. Although Renesas Electronics endeavours to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
13. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

    Note 1 “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority- owned subsidiaries.

    Note 2 “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

Regional information

Some information contained in this document may vary from country to country. Before using any Renesas Electronics product in your application, please contact the Renesas Electronics office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

Visit

http://www.renesas.com

to get in contact with your regional representatives and distributors.

Preface

Readers This manual is intended for users who want to understand the functions of the concerned libraries.

Purpose This manual presents the software manual for the concerned libraries.

Numeric notation

Binary:	xxxx or xxxB
Decimal:	xxxx
Hexadecimal	xxxxH or 0x xxxx

Numeric prefix Representing powers of 2 (address space, memory capacity):

K (kilo)	210 = 1024
M (mega):	220 = 1024^2 = 1,048,576
G (giga):	230 = 1024^3 = 1,073,741,824

Register X, x = don’t care

Diagrams Block diagrams do not necessarily show the exact software flow but the functional structure. Timing diagrams are for functional explanation purposes only, without any relevance to the real hardware implementation.

How to Use This Document

(1) Purpose and Target Readers

This manual is designed to provide the user with an understanding of the hardware functions and electrical characteristics of the MCU. It is intended for users designing application systems incorporating the MCU. A basic knowledge of electric circuits, logical circuits, and MCUs is necessary in order to use this manual. The manual comprises an overview of the product; descriptions of the CPU, system control functions, peripheral functions, and electrical characteristics; and usage notes.

Particular attention should be paid to the precautionary notes when using the manual. These notes occur within the body of the text, at the end of each section, and in the Usage Notes section.

The revision history summarizes the locations of revisions and additions. It does not list all revisions. Refer to the text of the manual for details.

(2) Related documents

Document number	Description
R01AN1416EDxxxx	EEPROM Emulation Library Type T02 (Tiny), European Release

(3) List of Abbreviations and Acronyms

Abbreviation	Full form
Block	Smallest erasable unit of a flash macro
Code Flash	Embedded Flash where the application code is stored. For devices without Data Flash EEPROM emulation might be implemented on that flash in the so called data area.
Data Flash	Embedded Flash where mainly the data of the EEPROM emulation are stored. Beside that also code operation might be possible.
Dual Operation	Dual operation is the capability to fetch code during reprogramming of the flash memory. Current limitation is that dual operation is only available between different flash macros. Within the same flash macro it is not possible!
EEL	EEPROM Emulation Library
EEPROM emulation	In distinction to a real EEPROM the EEPROM emulation uses some portion of the flash memory to emulate the EEPROM behavior. To gain a similar behavior some side parameters have to be taken in account.
FDL	Data Flash Library (Data Flash access layer)
Flash	“Flash EPROM” - Electrically erasable and programmable nonvolatile memory. The difference to ROM is, that this type of memory can be re-programmed several times.
Flash Block	A flash block is the smallest erasable unit of the flash memory.

Abbreviation	Full form
Flash Macro	A flash comprises of the cell array, the sense amplifier and the charge pump (CP). For address decoding and access some additional logic is needed.
NVM	Non volatile memory. All memories that hold the value, even when the power is cut off. E.g. Flash memory, EEPROM, MRAM...
RAM	“Random access memory” - volatile memory with random access
ROM	“Read only memory” - nonvolatile memory. The content of that memory can not be changed.
Serial programming	The onboard programming mode is used to program the device with an external programmer tool.
Single Voltage	For the reprogramming of single voltage flashes the voltage needed for erasing and programming are generated onboard of the microcontroller. No external voltage needed like for dual- voltage flash types.

All trademarks and registered trademarks are the property of their respective owners.

Table of Contents

Chapter 1 Introduction 10
1.1 Components of the EEPROM Emulation System 11
1.1.1 Physical Flash Layer 11
1.1.2 Flash Access Layer 11
1.1.3 EEPROM Access Layer 11
1.1.4 Application Layer 11
Chapter 2 Architecture 12
2.1 Data Flash Fragmentation 12
2.1.1 EEL Pool 12
2.1.2 FDL Pool 12
2.2 Address Virtualization 12
2.3 Access Right Supervision 13
2.4 Request-Response Architecture 15
2.5 Background Operation 16
2.5.1 Background Operation: Erase 16
2.5.2 Background Operation: Internal-Verify and Blankcheck 17
2.5.3 Background Operation (Write) 18
2.6 Abortion of Commands 19
2.7 StandBy and WakeUp Functionality 21
Chapter 3 User Interface (API) 22
3.1 Run-time Configuration 22
3.2 Data Types 23
3.2.1 Library-specific simple Type Definitions 23
3.2.2 Enumeration Type “fdl_command_t” 23
3.2.3 Enumeration Type “fdl_status_t” 24
3.2.4 Structured Type “fdl_request_t” 25
3.2.5 Structured Type “fdl_descriptor_t” 26
3.3 Functions 28
3.3.1 FDL_Init 28
3.3.2 FDL_Open 30
3.3.3 FDL_Close 32
3.3.4 FDL_Execute 34
3.3.5 FDL_Handler 36
3.3.6 FDL_Abort 38
3.3.7 FDL_StandBy 41

3.3.8 FDL_WakeUp 43
3.3.9 FDL_GetVersionString 45
3.4 Commands 47
3.4.1 Blankcheck 47
3.4.2 Internal Verify 48
3.4.3 Read 49
3.4.4 Write 50
3.4.5 Erase 51
3.5 Basic functional Workflow 52
Chapter 4 FDL Usage by User Application 53
4.1 First Steps 53
4.2 Special Considerations 53
4.2.1 Reset Consistency 53
4.2.2 EEL+FDL or FDL only 53
4.3 File Structure 54
4.4 Configuration 55
4.4.1 Linker Sections 55
4.4.2 Descriptor Configuration (Partitioning of the Data Flash) 55
4.4.3 Prohibited RAM Area 55
4.4.4 Register Bank 55
4.4.5 Stack and Data Buffer 56
4.4.6 Request Structure 56
4.5 General Flow 56
4.5.1 Initialization 56
4.5.2 Read 57
4.5.3 Blankcheck/Write/IVerify/Erase 58
4.6 Example of FDL used in Operating Systems 59
4.7 Example: Simple application 61
Chapter 5 Characteristics 62
5.1 Resource Consumption 62
5.2 Library Timings 62
5.2.1 Maximum Function Execution Times 62
5.2.2 Command Execution Times 64
Chapter 6 Cautions 65

Chapter 1 Introduction

This user's manual describes the overall structure, functionality and software interfaces (API) of the Data Flash Library T02 (Tiny) accessing the physical Data Flash separated and independent from the Code Flash. This library supports dual operation mode where the content of the Data Flash is accessible (read, write, erase) during instruction code execution.

The flash access layer is a layer of the EEPROM emulation system and encapsulates the low-level access to the physical flash a secure way. In case of Data Flash, this layer is using the FDL. It provides a functional socket for Renesas EEPROM emulation software.

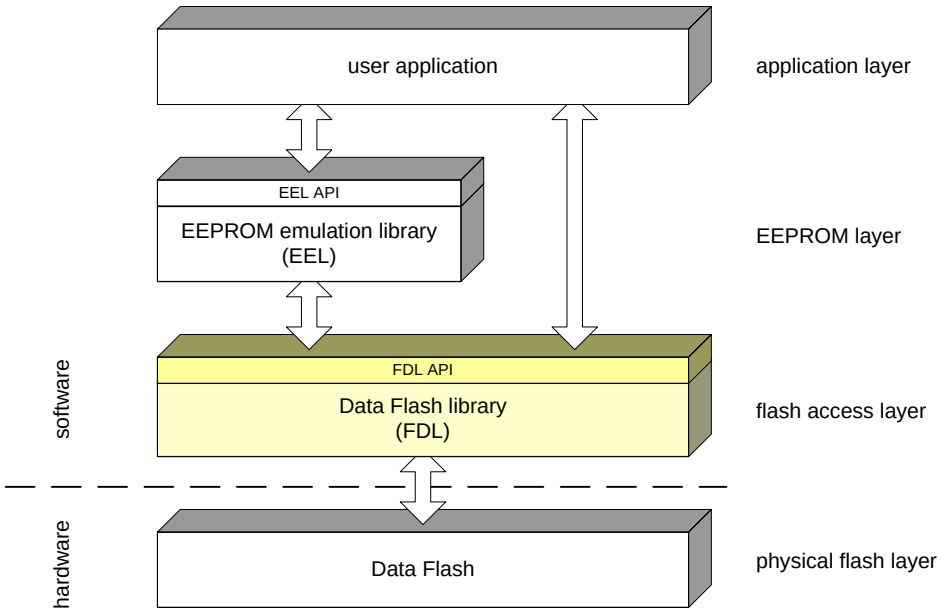

Figure 1: Components of the EEPROM emulation system

To boost the flexibility and the real-time characteristics of the library it offers only fast atomic functionality to read, write and erase the Data Flash memory at smallest possible granularity. Beside the pure access commands some maintenance functionality to check the quality of the flash content is also provided by the library.

1.1 Components of the EEPROM Emulation System

To achieve a high degree of encapsulation the EEPROM emulation system is divided into several layers with narrow functional interfaces.

1.1.1 Physical Flash Layer

The FDL is accessing the Data Flash as a physical media for storing data in the EEPROM emulation system. The Data Flash is a separate memory that can be accessed independent of the Code Flash memory. This allows background access to data stored in the Data Flash during program execution located in the code flash.

1.1.2 Flash Access Layer

The flash access layer is represented by the flash access library provided by Renesas. In case of devices incorporating data-flash the Data Flash Library (FDL) is representing this layer. It offers all atomic functionality to access the Data Flash. To isolate the data-flash access from the used flash-media this layer (the FDL) is transforming the physical addresses into a virtual, linear address-room.

1.1.3 EEPROM Access Layer

The EEPROM layer allows read/write access to the Data Flash on an abstract level. It is represented by a Renesas EEPROM Emulation Library (EEL) or alternatively any other, user specific implementation.

1.1.4 Application Layer

The application layer represents user's application software that can freely access all visible (specified by the API definition) commands. The EEPROM layer and the flash access layer can be used simultaneously. The FDL manages the access rights to it in a proper way.

Chapter 2 Architecture

This chapter describes the overall architecture of the Tiny FDL.

2.1 Data Flash Fragmentation

The physical address range of the Data Flash depends on the utilized hardware (e.g. for RL78/G13: 0xF1000 – 0xF1FFF). However, the logical fragmentation of the Data Flash can be configured within the given range.

Following figure shows the logical fragmentation of RL78/G13 physical Data Flash.

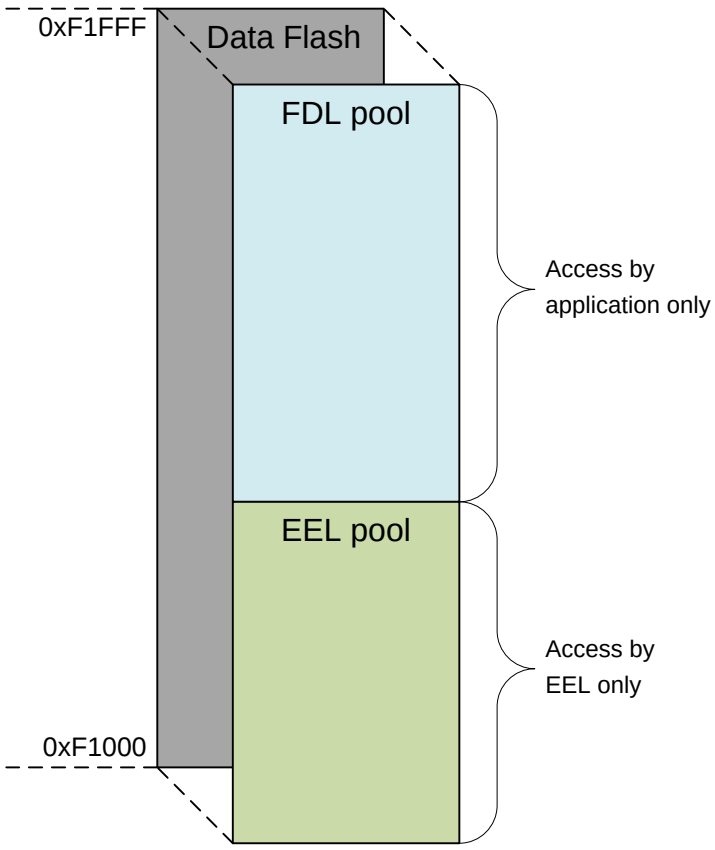

Figure 2: Logical fragmentation of physical Data Flash

2.1.1 EEL Pool

The EEL pool is exclusively used by the Renesas EEPROM Emulation Library (EEL). In case the EEL is not used the whole Data Flash can be reserved as FDL pool.

2.1.2 FDL Pool

The FDL pool is exclusively used by the application. In case of a proprietary EEPROM emulation implementation (user specific), the complete FDL pool has to be configured as FDL-pool.

2.2 Address Virtualization

To facilitate the access to the FDL pool, the physical addresses were virtualized. The virtualized pool looks like a simple one-dimensional array.

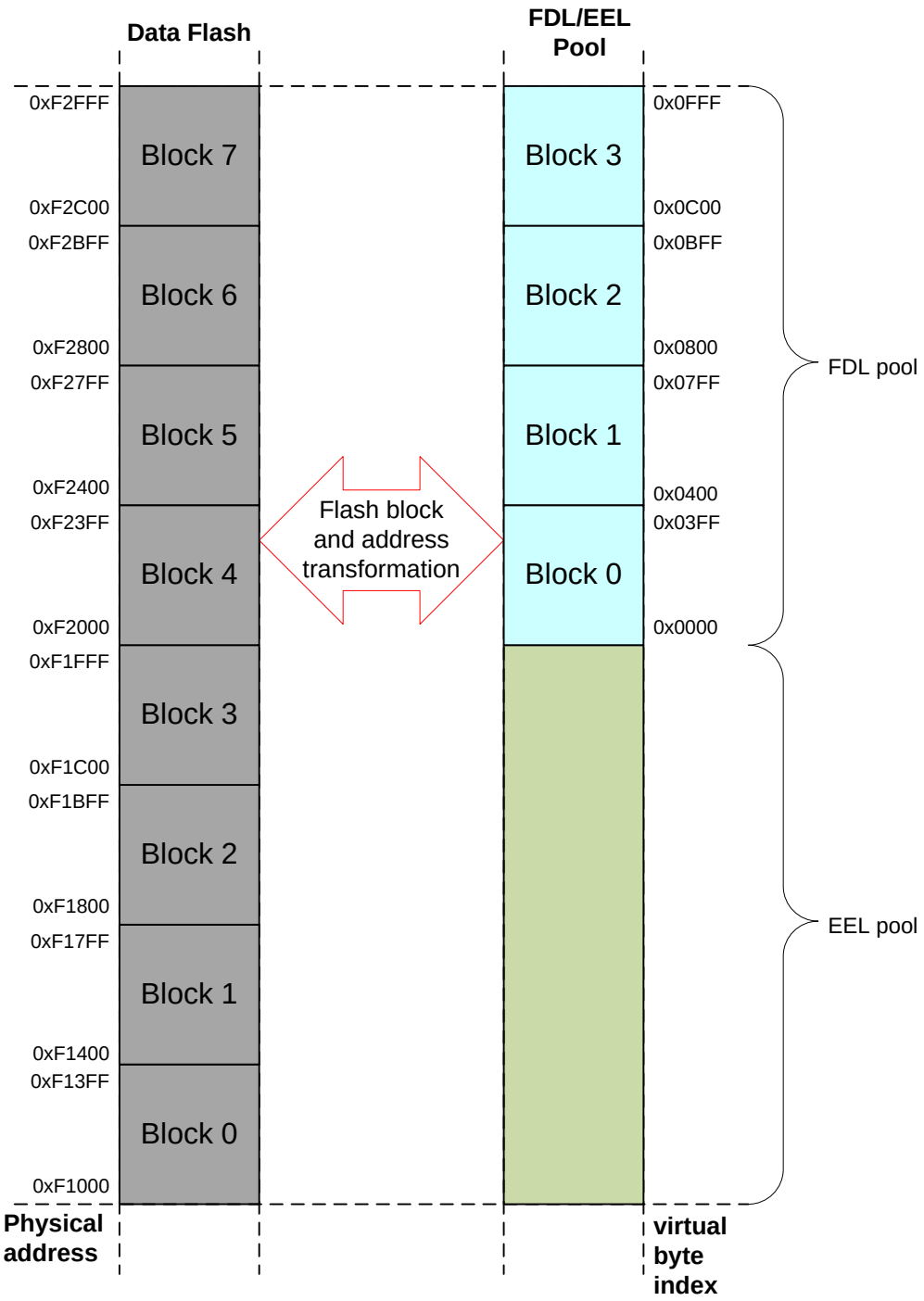

Figure 3: Relationship between physical and virtual pool addresses

2.3 Access Right Supervision

As mentioned before, the complete Data Flash is divided into two parts which are accessable by pool owner (FDL does not allow user access to the EEL-pool and vice versa).

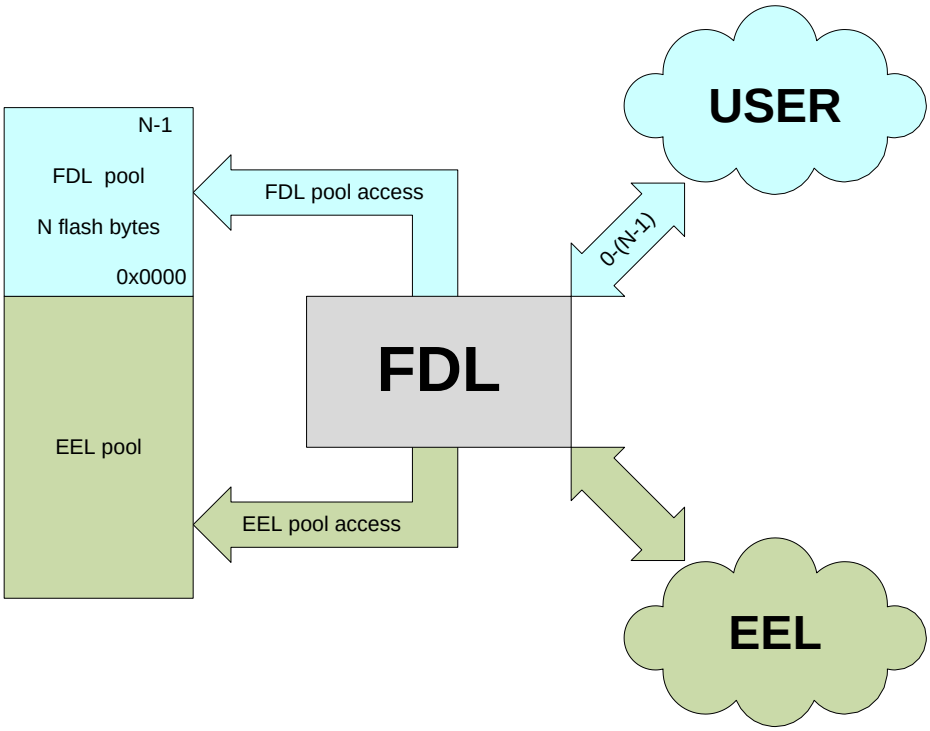

Figure 4: FDL pool access supervision

2.4 Request-Response Architecture

The communication between the requester (user) and the executor (here the FDL) is a structured request variable. The requester can specify the request and pass it to the FDL. After acceptance, the progress of the execution can be checked by polling the request status.

From execution-time point of view the commands of the FDL are divided into two groups:

- commands that can be aborted: block oriented commands like erase taking relatively long time for its execution
- commands that cannot be aborted: byte-oriented commands like write, read ... taking very short time for its execution

Depending on the real-time requirements, the user can decide if independent, quasi-parallel execution of block and byte commands is required or not. In such a case, two separate request-variables have to be defined and managed by the application. Please refer to chapter “Basic functional Workflow” for details.

Following figure shows the access from requester and FDL point of view.

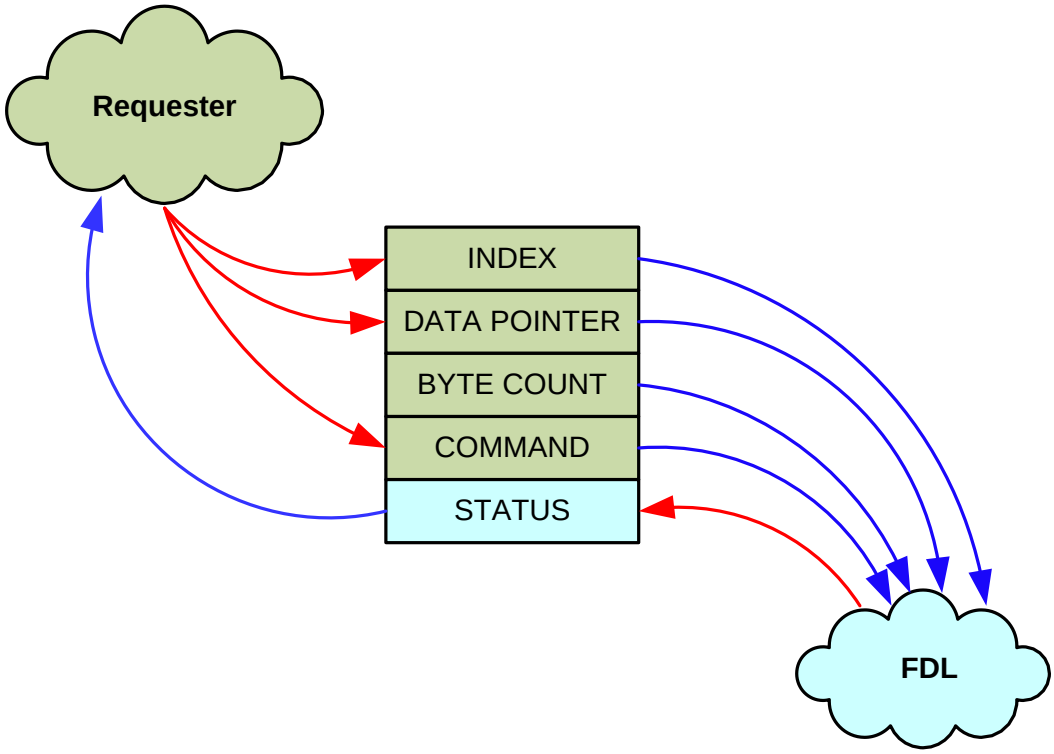

Figure 5: Request oriented communication between FDL and its requester

2.5 Background Operation

The flash technology provided by Renesas enables the application to write/erase the Data Flash in parallel to the CPU execution. Such a feature is a powerful especially in operation systems were each task could start FDL commands which will be executed in the background during task switching.

2.5.1 Background Operation: Erase

The erase command is the longest command. Once started, the erase command is executed in the background leaving the user the option to execute other application tasks in the meantime. By calling the FDL_Handler, the current progress of the command can be checked via the status of the used request structure. As shown in the figure below, the application has the possibility to execute other user code during the background operation.

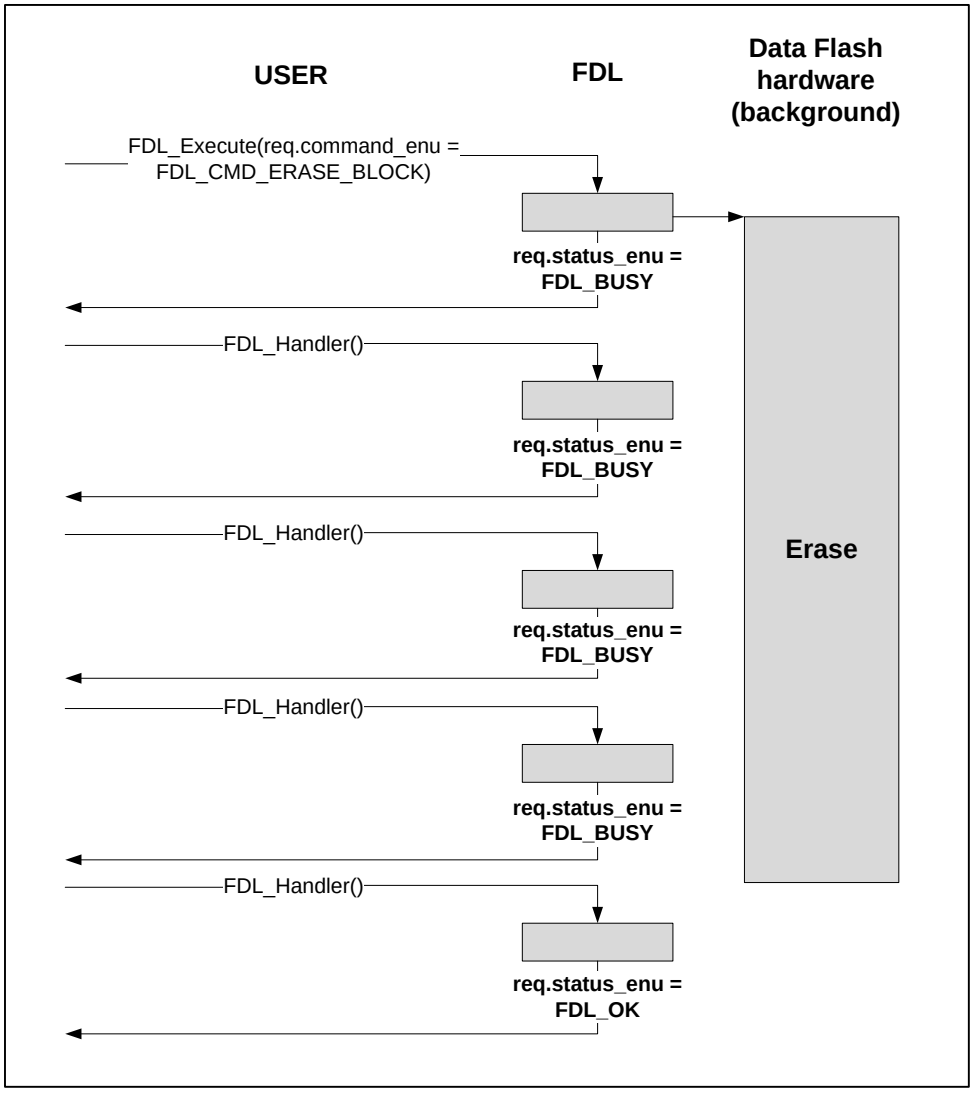

Figure 6: Background operation (Erase)

2.5.2 Background Operation: Internal-Verify and Blankcheck

Just like the erase command, also the internal verify and the blankcheck are performed in the background, once they have been triggered.

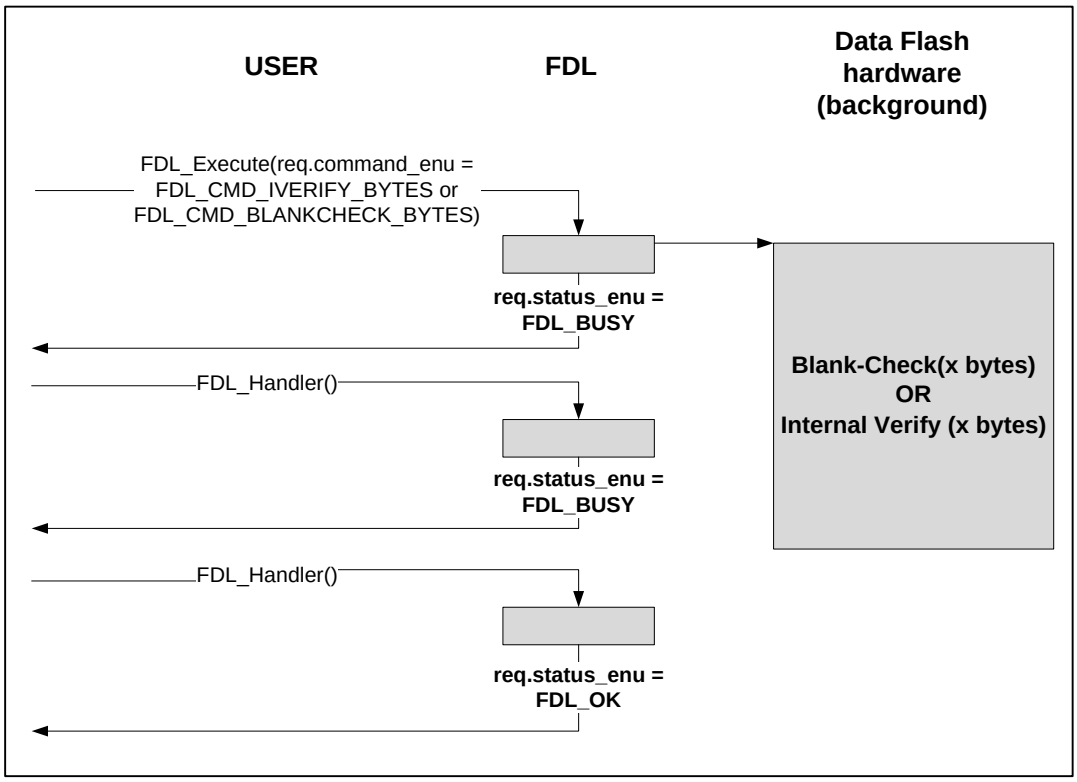

Figure 7: Background operation (Internal-Verify/Blankcheck)

2.5.3 Background Operation (Write)

Compared to the erase/verify/blankcheck command the write commands, is running only partially in the background. Each byte is written in the background whereas the administrative part of selecting the next byte is done by the FDL_Handler(). Therefore, it is mandatory to call the FDL_Handler not only for checking the current progress, but also to drive the command forward.

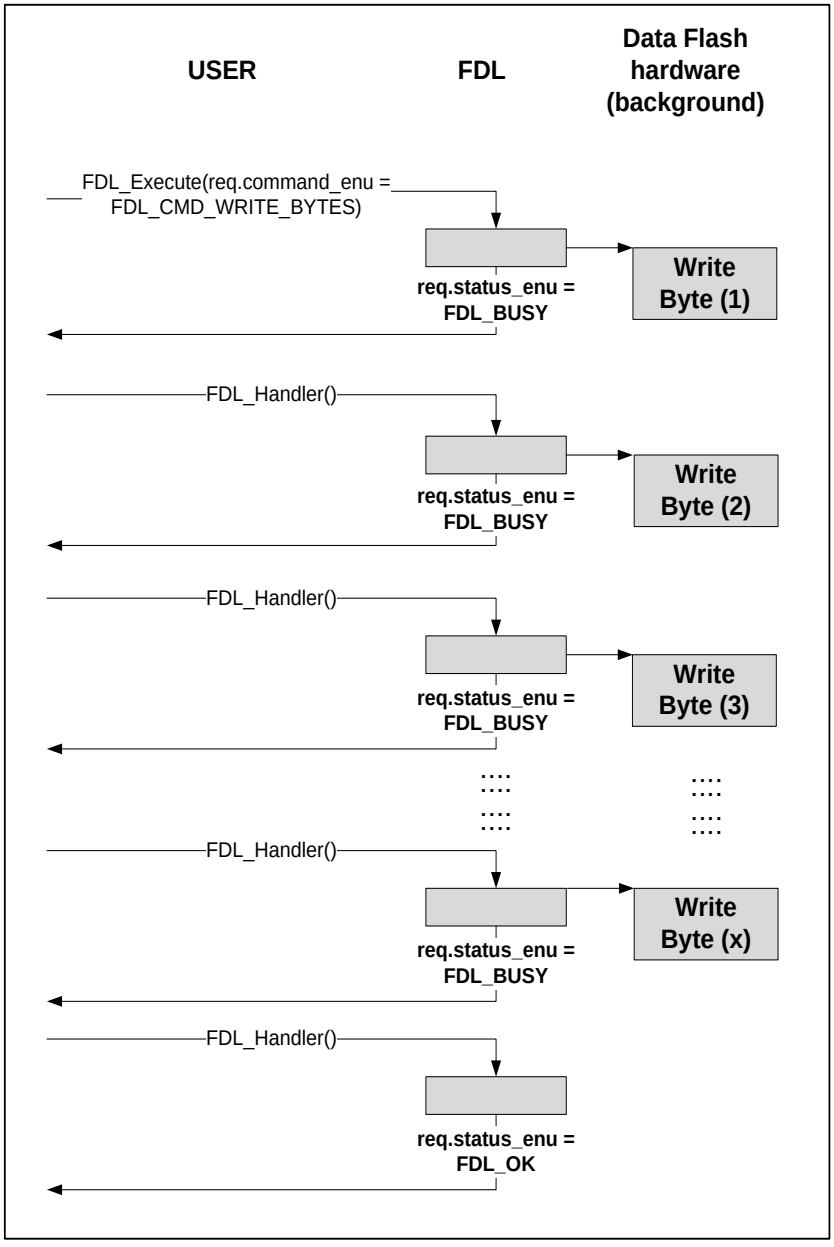

Figure 8: Background operation (Write)

2.6 Abortion of Commands

Some application scenarios require an immediate abort of running data flash operations e.g. in cases of voltage drop or emergency data write to the data flash. For that reason the FDL provides an abort mechanism where a running erase command can be aborted immediately. The following figure shows such an scenario.

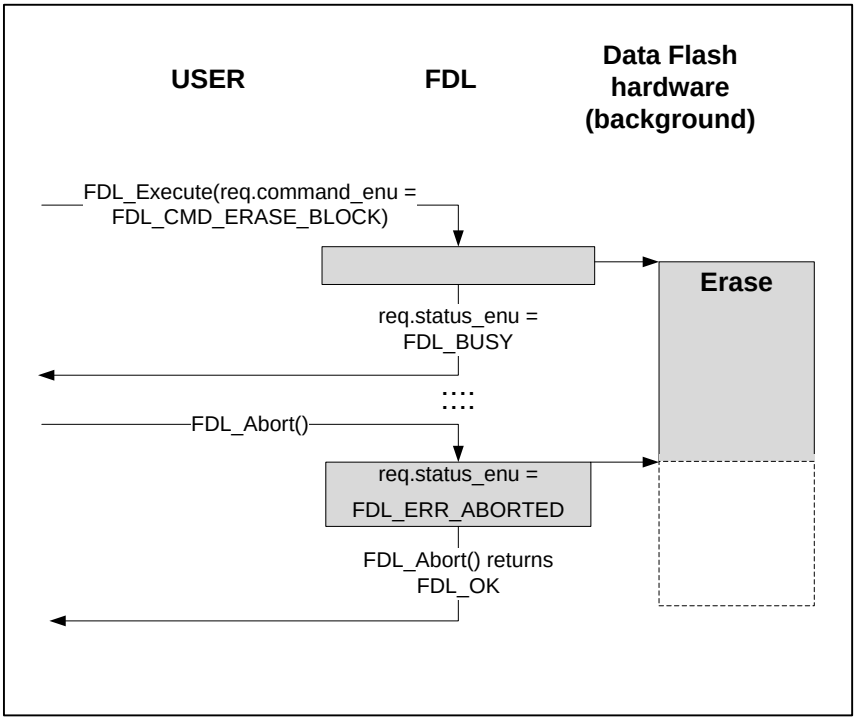

Figure 9: Abort erase command

As shown in the figure above, the erase command will be immediately aborted in case of calling the FDL_Abort function and the requester of the erase command will be informed that the requested command was aborted. In such a case, the command shall be re-started by requester later. In contrast, the blankcheck/write/verify commands cannot be aborted immediately and therefore have to be finished by repeated calls of the FDL_Abort function. The following figure shows the abort functions in case of blankcheck/write/verify commands.

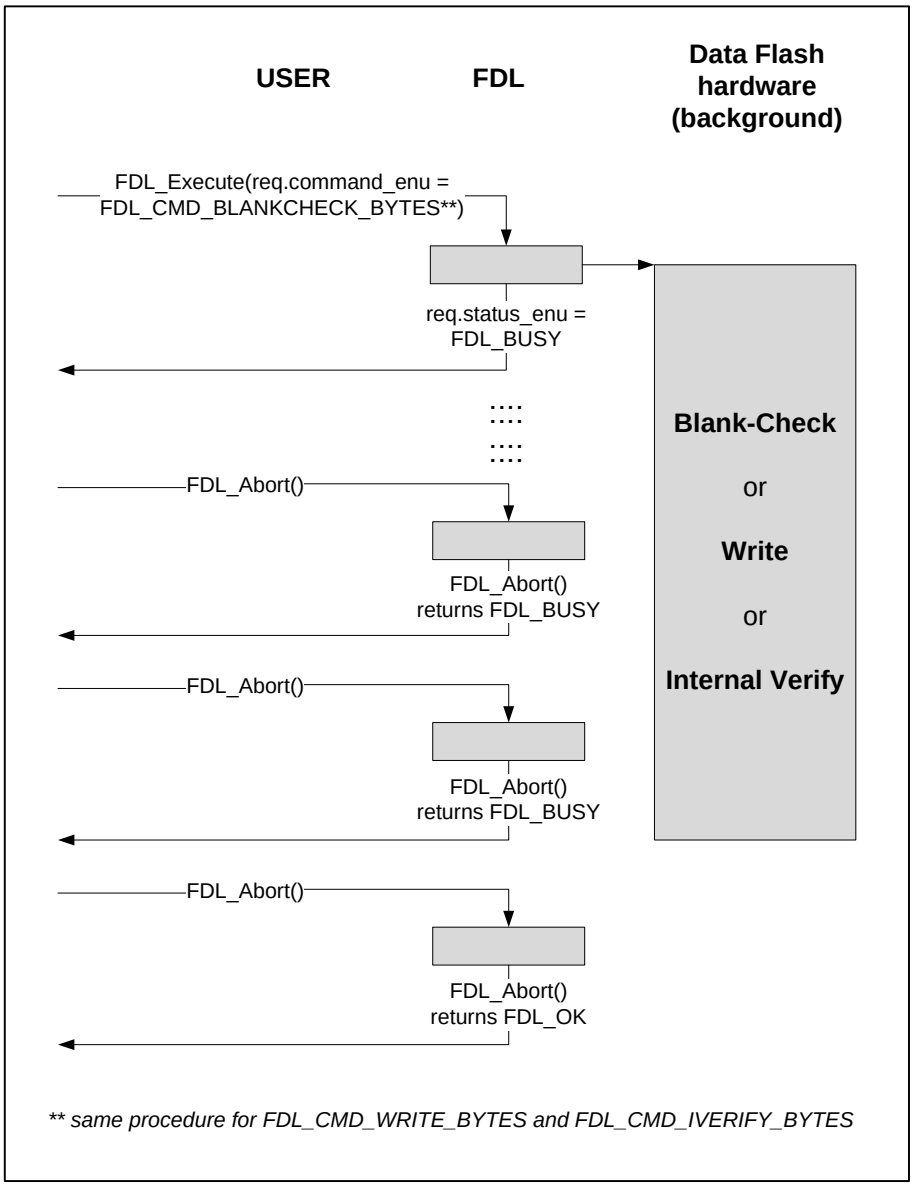

Figure 10: Abort Blankcheck/Write/Internal Verify command

2.7 StandBy and WakeUp Functionality

The StandBy and WakeUp feature provides the possibility to temporarily turn off the data flash functionality including the hardware (e.g. for power consumption) and resume the functionality. The StandBy mode is active in case the FDL_StandBy function returns FDL_OK. In case the return status is FDL_BUSY any command is running and FDL_StandBy function has to be re-called. The following figure shows the sequence of using the StandBy/WakeUp feature.

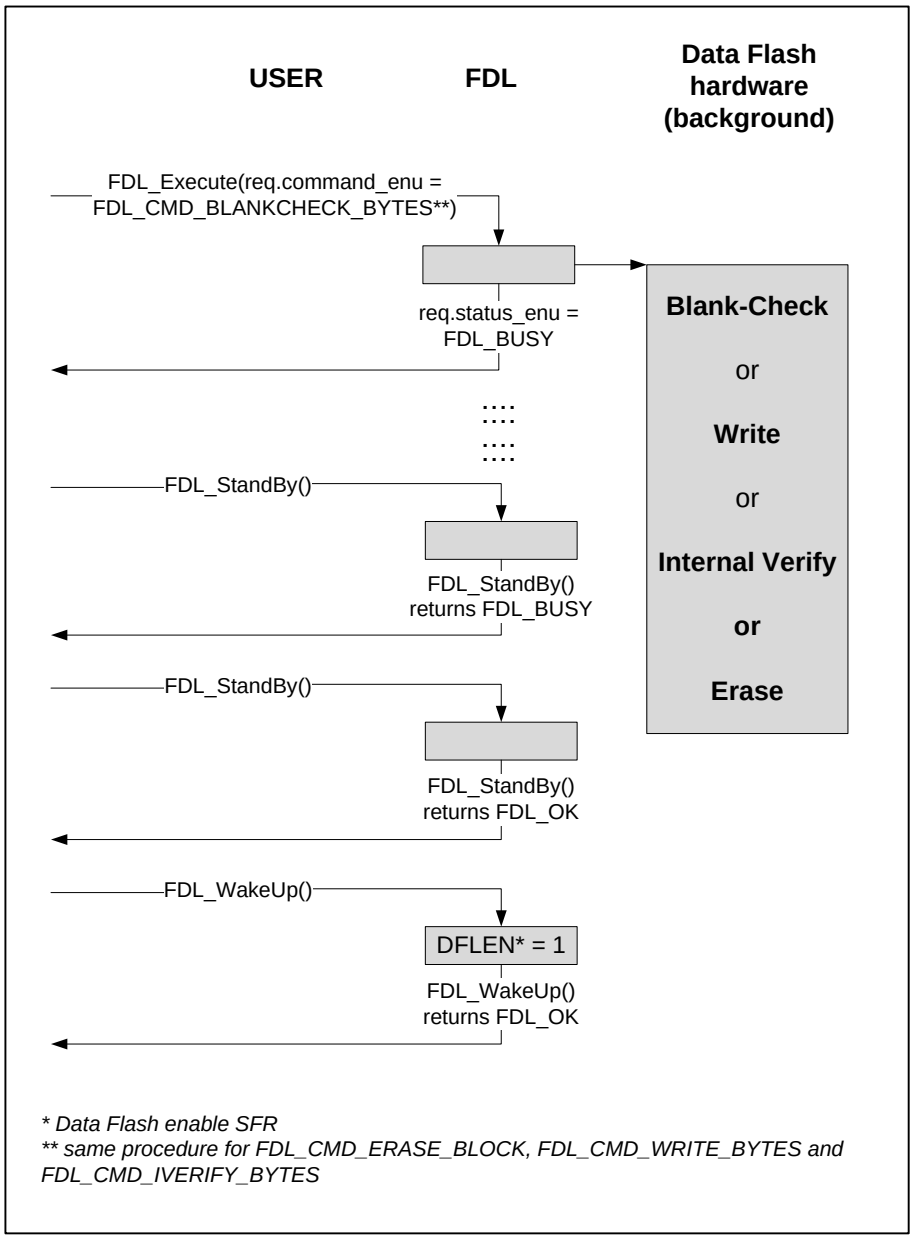

Figure 11: StandBy and WakeUp sequence

Note: It is not allowed to call any FDL function other than FDL_WakeUp and FDL_Handler when FDL is in StandBy mode.

Chapter 3 User Interface (API)

3.1 Run-time Configuration

The configuration of the FDL can be changed dynamically at runtime. Thereby, more than one descriptor constant has to be defined by the user in advance. Depending on the application different descriptors can be used for the FDL_Init(...) function.

```
/*    ..................... */
/*    some code             */
/*    ..................... */

/* load standard descriptor */
my_status=FDL_Init(&fdl_descriptor_str);

/*    ..................... */
/*    some code             */
/*    ..................... */

EEL_Close();
FDL_Close();


/* load alternative descriptor     */
my_status=FDL_Init(&fdl_descr_2_str);

/*    ..................... */
/*    some code             */
/*    ..................... */
```

Note: Before changing FDL pool configuration by using a different FDL pool-descriptor, the user has to finish all running FDL and EEL commands and close the FDL via the FDL_Close() function.

3.2 Data Types

This chapter describes all data definitions used by the TinyFDL. In order to reduce the probability of type mismatches in the user application, please make strict usage of the provided types and avoid using standard data types instead.

3.2.1 Library-specific simple Type Definitions

This type defines simple numerical type used by the library

```
typedef unsigned char                    fdl_u08;
typedef unsigned int                     fdl_u16;
typedef unsigned long int                fdl_u32;
```

3.2.2 Enumeration Type “fdl_command_t”

The enumeration type fdl_command_t defines all allowed codes used to specify library commands. This type is used within the structure fdl_request_t (see Section 3.2.4) in order to specify which command shall be executed via the function FDL_Execute. A detailed description of each command can be found in Section 3.4.

```
typedef enum
{
       FDL_CMD_READ_BYTES          = (0x00),
       FDL_CMD_IVERIFY_BYTES       = (0x01),
       FDL_CMD_BLANKCHECK_BYTES    = (0x02),
       FDL_CMD_WRITE_BYTES         = (0x03),
       FDL_CMD_ERASE_BLOCK         = (0x04)
} fdl_command_t;
```

Note: Due to the fact that the library has been implemented in Assembler, it is mandatory that the enumeration type fdl_command_t has a size of exactly 1 byte. The GNU compiler uses 16-bit enumeration types by default. Therefore, for GNU compiler, the declaration of the enumeration type has to be extended with an attribute in order to be compiled to 1 byte:
“__attribute__ ((__packed__))”.

Table 1: Command codes

Command	Description
FDL_CMD_READ_BYTES	reads data from flash memory
FDL_CMD_IVERIFY_BYTES	verifies data if flash provides full data retention
FDL_CMD_BLANKCHECK_BYTES	checks if flash content is erased
FDL_CMD_WRITE_BYTES	writes data into flash memory
FDL_CMD_ERASE_BLOCK	erases one flash block

3.2.3 Enumeration Type “fdl_status_t”

This enumeration type defines all possible status- and error-codes can be generated during data-flash access via the FDL. The FDL_OK and FDL_BUSY status are returned to the requester during normal operation. Other codes signalize problems.

On the one hand, fdl_status_t is used as return type of the functions FDL_Init (see Section 3.3.1), FDL_Abort (see Section 3.3.6), FDL_StandBy (see Section 3.3.7) and FDL_WakeUp (see Section 3.3.8). On the other hand, fdl_status_t is used within the structure fdl_request_t (see Section 3.2.4) in order to capture the processing of currently running command. Thereby, the possible error codes are command specific and described in detail in Section 3.4 along with the commands.

```
typedef enum
{
        FDL_OK                      = (0x00),
        FDL_BUSY                    = (0xFF),
        FDL_ERR_INITIALIZATION      = (0x02),
        FDL_ERR_REJECTED            = (0x03),
        FDL_ERR_ABORTED             = (0x04),
        FDL_ERR_PARAMETER           = (0x05),
        FDL_ERR_STANDBY             = (0x06),
        FDL_ERR_ERASE               = (0x1A),
        FDL_ERR_BLANK_VERIFY        = (0x1B),
        FDL_ERR_WRITE               = (0x1C),
        FDL_ERR_CONFIGURATION       = (0x01)
} fdl_status_t;
```

Note: Due to the fact that the library has been implemented in Assembler, it is mandatory that the enumeration type fdl_status_t has a size of exactly 1 byte. The GNU compiler uses 16-bit enumeration types by default. Therefore, for GNU compiler, the declaration of the enumeration type has to be extended with an attribute in order to be compiled to 1 byte:
“__attribute__ ((__packed__))”.

Table 2: Enumeration type "fdl_status_t" details

Status value	Description
FDL_OK	Command finished without problems
FDL_BUSY	Command is being processed
FDL_ERR_INITIALIZATION	FDL_Init()/FDL_Open() missing
FDL_ERR_REJECTED	Request could not be accepted
FDL_ERR_ABORTED	Erase command has been aborted
FDL_ERR_PARAMETER	Parameter error
FDL_ERR_STANDBY	FDL_WakeUp missing
FDL_ERR_ERASE	Erase error
FDL_ERR_BLANK_VERIFY	Blank- or verify error
FDL_ERR_WRITE	Write error
FDL_ERR_CONFIGURATION	Pool or frequency configuration wrong

3.2.4 Structured Type “fdl_request_t”

This type is used for definition of request variables and used for information exchange between the application and the FDL. A request variable is passed to the FDL to initiate a command and can be used by the requester (EEL, application...) to check the status of its execution. Not every element of this structure is required for each command. However, all members of the request variable must be initialized once before usage. Please refer to Section 3.4 for a more detailed description and the command-specific usage of the structure elements.

```
/* FDL request type (base type for any FDL access) */
typedef struct
{
       fdl_u16             index_u16;
       _near fdl_u08*      data_pu08;
       fdl_u16             bytecount_u16;
       fdl_command_t       command_enu;
       fdl_status_t        status_enu;
} fdl_request_t;
```

Note: The GNU compiler does not require the “__near” keyword to declare near pointers. All pointers are near by default as long as the “__far” keyword is not used.

Table 3: Structured type "fdl_request_t" details

Struct member	Description
index_u16	Start address of the target area: • Erase: virtual block number inside FDL-pool • Read/write/blankcheck/internal verify: virtual byte number inside FDL-pool
data_pu08	Pointer to the first byte of the data buffer to be written or read. Only used for read/write commands.
bytecount_u16	Number of bytes to be transferred starting from the start byte specified in index_u16. The byte count range is from 1 byte to 1024 bytes. Please note, that the execution of the read/write/blankcheck/internal verify command across block boundaries is not allowed. This struct member is not required for erase command.
command_enu	Command code to be executed
status_enu	Request status code (feedback)

3.2.5 Structured Type “fdl_descriptor_t”

This type defines the structure of the FDL descriptor. It contains all characteristics of the FDL. It is used in the fdl_descriptor.c sample file for definition of the ROM constant fdl_descriptor_str.

Based on configuration data inside the fdl_descriptor.h the initialization data of descriptor constant is generated automatically in the fdl_descriptor.c.

```
/* FDL descriptor type */
typedef struct
{
       fdl_u16     eel_pool_bytes_u16;
       fdl_u16     fdl_pool_bytes_u16;
       fdl_u16     fdl_delay_u16;
       fdl_u08     eel_pool_blocks_u08;
       fdl_u08     fdl_pool_blocks_u08;
       fdl_u08     fx_MHz_u08;
       fdl_u08     wide_voltage_mode_u08;
} fdl_descriptor_t;
```

Table 4: Structured type "fdl_descriptor_t" details

Struct member	Description
eel_pool_bytes_u16	EEL-pool size in bytes. It shall be computed by the compiler pre-processor based on the following formula: *DATA_FLASH_BLOCK_SIZE* eel_pool_blocks_u08* DATA_FLASH_BLOCK_SIZE is the physical size of a Flash block specified in the device HW user manual.
fdl_pool_bytes_u16	FDL-pool size in bytes It shall be computed by the compiler pre-processor based on the following formula: *DATA_FLASH_BLOCK_SIZE* eel_pool_blocks_u08* DATA_FLASH_BLOCK_SIZE is the physical size of a Flash block specified in the device HW user manual.
fdl_delay_u16	Constant delay depending on configured frequency. It shall be computed by the compiler pre-processor by the following formula: *(10 * fx_MHz_u08) / 6*
eel_pool_blocks_u08	EEL-pool size in blocks. It shall be chosen by the user with the condition that (fdl_pool_blocks_u08 + eel_pool_blocks_u08) may not exceed the size of the data flash specified in the device HW user manual.
fdl_pool_blocks_u08	FDL-pool size in blocks. It shall be chosen by the user with the condition that (fdl_pool_blocks_u08 + eel_pool_blocks_u08) may not exceed the size of the data flash specified in the device HW user manual.

Struct member	Description
fx_MHz_u08	CPU frequency Frequency must be rounded up as follows: descr.fx_MHz_u08 = *((FDL_SYSTEM_FREQUENCY+999999)/1000000)* FDL_SYSTEM_FREQUENCY specifies the device frequency and not the HOCO (internal high-speed on-chip oscillator) frequency. In case the frequency is smaller than 4MHz the only supported physically frequencies by FDL are the following: 1MHz=1000000Hz, 2MHz=2000000Hz and 3MHz=3000000Hz
wide_voltage_mode_u08	Programming voltage mode (full/wide). It shall be chosen by the user: • wide_voltage_mode_u08 shall be set to 1 for wide voltage mode • wide_voltage_mode_u08 shall be set to 0 for full speed voltage mode. For details of the voltage mode, refer to the user's manual of the target RL78 microcontroller.

3.3 Functions

3.3.1 FDL_Init

Outline: Initialization of the Flash Data Library.

Interface: **C Interface for CA78K0R Compiler**

```
fdl_status_t  __far FDL_Init(const __far fdl_descriptor_t*
                                                    descriptor_pstr)
```

C Interface for IAR Compiler

```
__far_func fdl_status_t FDL_Init(const __far fdl_descriptor_t __far*
                                                    descriptor_pstr)
```

C Interface for GNU Compiler

```
fdl_status_t  FDL_Init(const fdl_descriptor_t __far* descriptor_pstr)
                                __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
fdl_status_t __far FDL_Init(const __far fdl_descriptor_t*
                                                  descriptor_pstr);
```

ASM function label

```
FDL_Init
```

Arguments: **Parameters**

Argument	Type	Access	Passed via			
			CA78K0R	IAR	GNU	CC-RL
descriptor_pstr	fdl_descriptor_t* (far)	R	BC(highw), AX(loww)	stack	stack	A(high), DE(loww)
Pointer to the descriptor (describing the FDL configuration). The virtualization of the data-flash address-room is done based on that descriptor. The user can use different descriptors to switch between different FDL-pool configurations.						

Return value

Type	Passed via			
	CA78K0R	IAR	GNU	CC-RL
fdl_status_t	C	A	R8 (X bank 1)	A
FDL_ERR_CONFIGURATION when descriptor is not plausible. FDL_OK when descriptor is plausible and initialization was successful.				

Destructed registers

Tool chain	Destructed registers
CA78K0R	AX, B
IAR	AX, HL, CS, ES
GNU	None
CC-RL	X, BC, DE, HL

Pre-conditions: Internal high-speed oscillator is running.

Post-conditions: Initialization is done.

Description: Several checks are performed during the initialization:

- plausibility check of the pool configuration
- frequency parameter check against supported device-specific range
- initialization of all internal variables
- initialization of the flash firmware
- configuration of HOCO

After initialization, the FDL remains passive. FDL_Open() has to be executed to open access to the FDL pool.

Note: It is not allowed to call FDL_Init in case of any running FDL command.

Example:

```
fdl_status_t my_status;
my_status = FDL_Init(&fdl_descriptor_str);
if(my_status == FDL_OK)
{
/* FDL can be used */
}
else
{
/* error handler */
}
```

3.3.2 FDL_Open

Outline: Activation of the data-flash.

Interface: **C Interface for CA78K0R Compiler**

```
void __far FDL_Open(void)
```

C Interface for IAR Compiler

```
__far_func void FDL_Open(void)
```

C Interface for GNU Compiler

```
void  FDL_Open(void) __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
void __far FDL_Open(void)
```

ASM function label

```
FDL_Open
```

Arguments: **Parameters**

none

Return value

none

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	AX

Pre-conditions: The initialization shall be done before. However, no check is performed here. If the FDL is not yet initialized, FDL_Open() has no functionality.

Post-conditions: Data flash clock is switched on.

Description: This function must be used by the application to activate the data-flash.

Example:

```
FDL_Open();
```

3.3.3 FDL_Close

Outline: Deactivation of the data-flash.

Interface: **C Interface for CA78K0R Compiler**

```
void __far FDL_Close(void)
```

C Interface for IAR Compiler

```
__far_func void FDL_Close(void)
```

C Interface for GNU Compiler

```
void FDL_Close(void) __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
void __far FDL_Close(void)
```

ASM function label

```
FDL_Close
```

Arguments: **Parameters**

none

Return value

none

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	C

Pre-conditions: The library initialization and open via FDL_Init and FDL_Open shall be done before calling this function. If FDL is not yet activated the FDL_Close() has no functionality.

Post-conditions: Data flash clock is switched off. All hardware background activities will be stopped immediately.

Description: This function must be used by the application to deactivate the data-flash.

Example:

```
FDL_Close();
```

3.3.4 FDL_Execute

Outline: Initiates the execution of an FDL command.

Interface: **C Interface for CA78K0R Compiler**

```
void __far FDL_Execute(__near fdl_request_t* request_pstr)
```

C Interface for IAR Compiler

```
__far_func void FDL_Execute(__near fdl_request_t __near* request_pstr)
```

C Interface for GNU Compiler

```
void FDL_Execute(fdl_request_t* request_pstr)
                                    __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
void __far FDL_Execute(__near fdl_request_t* request_pstr)
```

ASM function label

```
FDL_Execute
```

Arguments: **Parameters**

Argument	Type	Access	Passed via			
			CA78K0R	IAR	GNU	CC-RL
request_pstr	fdl_request_t* (near)	RW	AX	AX	stack	AX
This argument defines the command which should be executed by FDL. It is a request variable which is used for bidirectional information exchange before and during execution between FDL and the application. For details please refer to the “Request-Response Architecture” chapter.						

Return value

none

Destructed registers

Tool chain	Destructed registers
CA78K0R	AX
IAR	AX, HL, CS, ES
GNU	None
CC-RL	AX, BC, DE, HL

Pre-conditions: The library initialization and open via FDL_Init and FDL_Open shall be done before calling this function.

Post-conditions: None

Description: This is the main function of the FDL which can be used inside of the application to initiate the execution of any command. Please refer to the chapter “Commands” for detailed explanation of each command.

Note 1: Although there are commands that do not require all request structure elements to be specified, the whole structure needs to be initialized before calling EEL_Execute. Otherwise, a RAM parity error may cause a reset of the device. For details, please refer to the document “User's Manual: Hardware” of your RL78 product.

Note 2: The request structure used for execution has to be word-aligned, i.e. located at an even memory address.

Example:

```
__near fdl_request_t my_fdl_request_str;
__near fdl_u08 buffer[4];

buffer[0] = {0x01, 0x23, 0x45, 0x67};

my_fdl_request.index_u16 = 0x0000;
my_fdl_request.data_pu08 = (__near fdl_u08*)buffer;
my_fdl_request.bytecount_u16 = 0x0004;
my_fdl_request.command_enu = FDL_CMD_WRITE_BYTES;

/* command initiation */
do
{
       FDL_Execute(&my_fdl_request);
       FDL_Handler(); /* proceed background process */
}
while (my_fdl_request.status_enu == FDL_ERR_REJECTED);

/* command execution */
do
{
       FDL_Handler();
}
while (my_fdl_request.status_enu == FDL_BUSY);
if(my_fdl_request.status_enu != FDL_OK)
{
       error_handler();
}
```

3.3.5 FDL_Handler

Outline: Function for command proceeding.

Interface: **C Interface for CA78K0R Compiler**

```
void __far FDL_Handler(void)
```

C Interface for IAR Compiler

```
__far_func void FDL_Handler(void)
```

C Interface for GNU Compiler

```
void FDL_Handler(void) __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
void __far FDL_Handler(void)
```

ASM function label

```
FDL_Handler
```

Arguments: **Parameters**

none

Return value

none

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	A, C, HL

Pre-conditions: The library initialization and open via FDL_Init and FDL_Open shall be done before calling this function.

Post-conditions: In case of finished command the status is written to the request structure associated to the currently running command.

Description: This function is used by the application to proceed the execution of a running command initiated by FDL_Execute function.

Example:

```
/* infinite scheduler loop */
do
{
       /* proceed potential command execution */
       FDL_Handler();

       /* 20ms time slize (potential FDL requester) */
       MyTask_A(20);

       /* 10ms time slize (potential FDL requester) */
       MyTask_B(10);

       /* 40ms time slize (potential FDL requester) */
       MyTask_C(40);

       /* 10ms time slize (potential FDL requester) */
       MyTask_D(10);
}
while (true);
```

3.3.6 FDL_Abort

Outline: Function for erase command abortion.

Interface: **C Interface for CA78K0R Compiler**

```
fdl_status_t __far FDL_Abort(void)
```

C Interface for IAR Compiler

```
__far_func fdl_status_t FDL_Abort(void)
```

C Interface for GNU Compiler

```
fdl_status_t FDL_Abort(void) __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
fdl_status_t __far FDL_Abort(void)
```

ASM function label

```
FDL_Abort
```

Arguments: **Parameters**

none

Return value

Type	Passed via			
	CA78K0R	IAR	GNU	CC-RL
fdl_status_t	C	A	R8 (X bank 1)	A
FDL_OK when either no command is running or erase has been aborted. FDL_BUSY when byte oriented command is still running.				

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	BC, HL

Pre-conditions: The library initialization and open via FDL_Init and FDL_Open shall be done before calling this function.

Post-conditions: In case of a running erase the command will be aborted and requester will be informed via the request status FDL_ERR_ABORTED.

Description: This function enables the application to abort a running erase command (independent of the affected pool) immediately. The requester will be informed regarding the stopped erase via the request status FDL_ERR_ABORTED. In such a case the application shall re-start the erase command otherwise the block cannot be used. Other commands like blankcheck, write and verify cannot be aborted and therefore have to be finished properly. If the application calls this function during the above described byte commands the return value FDL_BUSY will be returned. That means a byte command is still running. Please re-call the FDL_Abort functions as long as the status is FDL_BUSY. Only when the return value is changed to FDL_OK the command is properly finished.

Example:

```
__near fdl_request_t       my_fdl_request_str;
__near fdl_u08             cmd_finished = 0;

/* request structure initialization */
my_fdl_request.index_u16          = 0x0000;
my_fdl_request.data_pu08          = (__near fdl_u08*) 0x0000;
my_fdl_request.bytecount_u16      = 0x0000;
my_fdl_request.command_enu        = (fdl_command_t)0xFF;
my_fdl_request.status_enu         = FDL_ERR_PARAMETER;

/* set erase command */
my_fdl_request.index_u16          = 0x0000;
my_fdl_request.command_enu        = FDL_CMD_ERASE_BLOCK;

/* ################# TASK 1 ################### */
/* ######## TRY TO ERASE HERE ONE BLOCK ######## */
/* ############################################## */

FDL_Execute(&my_fdl_request);

cmd_finished = 0;
while (cmd_finished == 0)
{
       switch (my_fdl_request.status_enu)
       {
              case FDL_BUSY:
                     FDL_Handler();
                     break;
              case FDL_ERR_ABORTED:
                     /* start erase again in case it was aborted */
                     FDL_Execute(&my_fdl_request);
                     FDL_Handler();
                     break;
              case FDL_ERR_REJECTED:
                     /* try again if request not accepted */
                     FDL_Execute(&my_fdl_request);
                     FDL_Handler();
                     break;
              default:
                     cmd_finished = 1;
                     break;
       }
}
```

```
if (my_fdl_request.status_enu != FDL_OK)
{
       error_handler();
}

/* ################### TASK 2 ####################### */
/* ########  WILL BE USED FOR EMERGENCY WRITE ######## */
.....

do
{
       abort_status = FDL_Abort();
}
while( abort_status != FDL_OK);

DRIVER IS NO MORE BUSY HERE.
PERFORM EMERGENCY WRITE HERE ....................
```

3.3.7 FDL_StandBy

Outline: Function to drive the library into StandBy mode.

Interface: **C Interface for CA78K0R Compiler**

```
fdl_status_t __far FDL_StandBy(void)
```

C Interface for IAR Compiler

```
__far_func fdl_status_t FDL_StandBy(void)
```

C Interface for GNU Compiler

```
fdl_status_t FDL_StandBy(void) __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
fdl_status_t __far FDL_StandBy(void)
```

ASM function label

```
FDL_StandBy
```

Arguments: **Parameters**

none

Return value

Type	Passed via			
	CA78K0R	IAR	GNU	CC-RL
fdl_status_t	C	A	R8 (X bank 1)	A
FDL_OK when FDL entered StandBy mode. FDL_BUSY any command is still running.				

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	C, HL

Pre-conditions: The library initialization and open via FDL_Init and FDL_Open shall be done before calling this function.

Post-conditions: Data flash clock is switched off and library is in StandBy mode.

Description: The main purpose of this function is to drive the library and Data Flash into the StandBy mode. StandBy mode means that

- the Data Flash hardware is switched off (the DFLEN bit of the Data flash control register is cleared), and
- the library does not accept any command requests

Note: It is not allowed to call any FDL function other than FDL_WakeUp and FDL_Handler when FDL is in StandBy mode.

Example:

```
do
{
       standby_status = FDL_StandBy();
}
while (standby_status != FDL_OK);


########################################
E.G. ENTER STOP/HALT MODE HERE ........
########################################


wakeup_status = FDL_WakeUp();

if(wakeup_status != FDL_OK)
{
       flow_error_handler();
}

E.G. CONTINUE WITH LIBRARY/APPLICATION EXECUTION HERE ..............
```

3.3.8 FDL_WakeUp

Outline: Function to wake up the library from StandBy mode.

Interface: **C Interface for CA78K0R Compiler**

```
__far_func fdl_status_t FDL_WakeUp(void)
```

C Interface for IAR Compiler

```
fdl_status_t __far FDL_WakeUp(void)
```

C Interface for GNU Compiler

```
fdl_status_t FDL_WakeUp(void) __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
__far_func fdl_status_t FDL_WakeUp(void)
```

ASM function label

```
FDL_WakeUp
```

Arguments: **Parameters**

none

Return value

Type	Passed via			
	CA78K0R	IAR	GNU	CC-RL
fdl_status_t	C	A	R8 (X bank 1)	A
FDL_OK when FDL is up and running. FDL_ERR_REJECTED when library isn't in StandBy mode.				

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	X

Pre-conditions: The library initialization and open via FDL_Init and FDL_Open shall be done before calling this function.

Post-conditions: Data flash clock is switched on and library is up and running.

Description: The main purpose of this function is to wake-up the library and Data Flash hardware from the StandBy mode. After successful execution of this function,

- the Data Flash hardware is switched on(the DFLEN bit of the Data flash control register is set), and
- the FDL accepts new command requests.

Example:

```
wakeup_status = FDL_WakeUp();

if(wakeup_status != FDL_OK)
{
       flow_error_handler();
}

E.G. CONTINUE WITH LIBRARY EXECUTION HERE ..............
```

3.3.9 FDL_GetVersionString

Outline: Function for reading library version information.

Interface: **C Interface for CA78K0R Compiler**

```
__far fdl_u08* __far FDL_GetVersionString(void)
```

C Interface for IAR Compiler

```
__far_func fdl_u08 __far* FDL_GetVersionString(void)
```

C Interface for GNU Compiler

```
fdl_u08 __far* FDL_GetVersionString(void)
                                __attribute__ ((section ("FDL_CODE")))
```

C Interface for CC-RL Compiler

```
__far fdl_u08* __far FDL_GetVersionString(void)
```

ASM function label

```
FDL_GetVersionString
```

Arguments: **Parameters**

none

Return value

Type	Passed via			
	CA78K0R	IAR	GNU	CC-RL
fdl_u08* (far)	DE(highw), BC(loww)	A, HL	R8-R11 (AX, BC bank 1)	A(high), DE(loww)
Pointer to the first character of a zero terminated version string.				

Destructed registers

Tool chain	Destructed registers
CA78K0R	None
IAR	None
GNU	None
CC-RL	None

Pre-conditions: None

Post-conditions: None

Description: For version control at runtime the developer can use this function to find the starting character of the library version string.

The version string is a zero-terminated string constant that covers library-specific information and is based on the following structure: NMMMMTTTCCCCCXVVV..V, where:

- N : library type specifier (here 'D' for FDL)
- MMMM : series name of microcontroller (here 'RL78')
- TTT : type number (here T02)
- CCCCC : compiler information
  - 'Rxyy_' for CA78K0R compiler version x.yy
  - 'Ixyy_' for IAR compiler version x.yy
  - 'Uxxyy' for GNU compiler version xx.yy
  - 'Lxyyz' for CC-RL compiler version x.yy.0z
- X : memory/register models (here 'G' for general, i.e. all memory models)
- VVV..V : library version

Examples:

The version string of the Tiny FDL V1.00 for the CA78K0R compiler version 1.10 is: "DRL78T02R110_GV100"

The version string of the Tiny FDL V1.00 for the IAR compiler version 1.20 is: "DRL78T02I120_GV100"

The version string of the Tiny FDL V1.01 for the GNU compiler version 13.02 is: "DRL78T02U1302GV101"

The version string of the Tiny FDL V1.01 for the CC-RL compiler version 1.23.04 is: "DRL78T02L1234GV101"

Example:

```
my_version_string_pointer = FDL_GetVersionString();
```

3.4 Commands

3.4.1 Blankcheck

The blankcheck command can be used to check if all bits within the addressed range are still “erased” e.g. before initiating a write. The blankcheck command is initiated by FDL_Execute() and must be continued by FDL_Handler() as long as command is not finished (request status updated).

Note: Due to the fact that the blankcheck command execution across block boundaries is not allowed the byte count range vary between 1 byte up to 1024 bytes.

Table 5: Request variable usage for blankcheck command

index_u16	data_pu08	bytecount_u16	command_enu
byte index inside the FDL pool	unused	byte count (1 byte to 1024 bytes)	FDL_CMD_ BLANKCHECK_BYTES

Table 6: Status of FDL_CMD_BLANKCHECK_BYTES

Status	Class	Status meaning and handling	
FDL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	initialize and open FDL before using it
FDL_ERR_STANDBY	heavy	meaning	FDL is in standby and cannot accept new commands
		reason	wrong handling on user side
		remedy	call FDL_WakeUp() before initiating new commands
FDL_ERR_PARAMETER	heavy	meaning	request cannot be accepted
		reason	wrong command code, index outside the used pool or request data structure on odd address
		remedy	correct affected request member and try again
FDL_ERR_BLANK_VERIFY	light	meaning	at least one byte within the specified pool area is not “blank”
		reason	any bit in the addressed flash area is not erased
		remedy	nothing, free interpretation at requester side
FDL_ERR_REJECTED	normal	meaning	request cannot be accepted
		reason	other command is being executed
		remedy	call FDL_Handler() and try again
FDL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FDL_Handler() until status changes
FDL_OK	normal	meaning	request was finished regular
		reason	no problems during execution
		remedy	nothing

3.4.2 Internal Verify

The internal verify command can be used to check if all bits (0's and 1's) are electronically correct written. Inconsistent and weak data caused by an asynchronous RESET can be detected by using the internal verify command. The user can use the verify command freely to check the quality of user data. The internal verify command is initiated by FDL_Execute() and must be continued by FDL_Handler() as long as command is not finished (request-status updated).

Note: An execution of internal verify commands across block boundaries is not allowed. As a result the byte count can range from 1 byte up to 1024 byte.

Table 7: Request variable usage for internal verify command

index_u16	data_pu08	bytecount_u16	command_enu
byte index inside the FDL pool	unused	byte count (1 byte to 1024 bytes)	FDL_CMD_IVERIFY_ BYTES

Table 8: Status of FDL_CMD_IVERIFY_BYTES

Status	Class	Status meaning and handling	
FDL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	initialize and open FDL before using it.
FDL_ERR_STANDBY	heavy	meaning	FDL is in standby and cannot accept new commands
		reason	wrong handling on user side
		remedy	call FDL_WakeUp() before initiating new commands
FDL_ERR_PARAMETER	heavy	meaning	request cannot be accepted
		reason	wrong command code, index outside the used pool or request data structure on odd address
		remedy	correct affected request member and try again
FDL_ERR_BLANK_VERIFY	light	meaning	at least one byte within the specified pool area could not be verified
		reason	any bit in the addressed flash word is not electrically correct
		remedy	nothing, free interpretation at requester side
FDL_ERR_REJECTED	normal	meaning	request cannot be accepted
		reason	other command is being executed
		remedy	call FDL_Handler() and try again
FDL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FDL_Handler() until status changes
FDL_OK	normal	meaning	request was finished regularly
		reason	no problems during execution
		remedy	nothing

3.4.3 Read

The READ command can be used to read a number of bytes from a specific address range. It is initiated and finished directly by FDL_Execute(). FDL_Handler() is not needed in that case unless the FDL is busy with another command.

Note: An execution of read commands across block boundaries is not allowed. As a result the byte count can range from 1 byte up to 1024 byte.

Table 9: Request variable usage for read command

index_u16	data_pu08	bytecount_u16	command_enu
byte index inside the FDL pool	pointer to the read buffer	byte count (1 byte to 1024 bytes)	FDL_CMD_READ_BYTES

Table 10: Status of FDL_CMD_READ_BYTES

Status	Class	Status meaning and handling	
FDL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	initialize and open FDL before using it.
FDL_ERR_STANDBY	heavy	meaning	FDL is in standby and cannot accept new commands
		reason	wrong handling on user side
		remedy	call FDL_WakeUp() before initiating new commands
FDL_ERR_PARAMETER	heavy	meaning	request cannot be accepted
		reason	wrong command code, index outside the used pool or request data structure on odd address
		remedy	correct affected request member and try again
FDL_ERR_REJECTED	normal	meaning	request cannot be accepted
		reason	other command is being executed
		remedy	call FDL_Handler() and try again
FDL_OK	normal	meaning	request was finished regular
		reason	no problems during execution
		remedy	nothing

3.4.4 Write

The write command can be used for writing a number of bytes located in a RAM buffer to the data-flash. It is initiated by FDL_Execute() and must be continued by FDL_Handler() as long as command is not finished (request-status updated).

Note 1: An execution of write commands across block boundaries is not allowed. As a result the byte count can range from 1 byte up to 1024 byte.

Note 2: For a regular write, please follow the suggested sequence of blankcheck, write, internal verify in order to ensure full data retention.

Table 11: Request variable usage for write command

index_u16	data_pu08	bytecount_u16	command_enu
byte index inside the FDL pool	pointer to the write buffer	byte count (1 byte to 1024 bytes)	FDL_CMD_WRITE_ BYTES

Table 12: Status of FDL_CMD_WRITE_BYTES

Status	Class	Status meaning and handling	
FDL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	initialize and open FDL before using it.
FDL_ERR_STANDBY	heavy	meaning	FDL is in standby and cannot accept new commands
		reason	wrong handling on user side
		remedy	call FDL_WakeUp() before initiating new commands
FDL_ERR_PARAMETER	heavy	meaning	request cannot be accepted
		reason	wrong command code, index outside the used pool or request data structure on odd address
		remedy	correct affected request member and try again
FDL_ERR_WRITE	heavy	meaning	at least one byte within the specified pool area is not “blank”
		reason	any bit in the addressed flash word is not electrically correct
		remedy	nothing, free interpretation at requester side
FDL_ERR_REJECTED	normal	meaning	request cannot be accepted
		reason	other command is being executed
		remedy	call FDL_Handler() and try again
FDL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FDL_Handler() until status changes
FDL_OK	normal	meaning	request was finished regularly
		reason	no problems during execution
		remedy	nothing

3.4.5 Erase

The erase operation can be used to erase one block of the pool. After starting the erase-command, the hardware is checking if the addressed block is already blank to avoid unnecessary erase cycles. In case the block is not blank the erase pulse is initiated, otherwise the erase command will be finished immediately**.**

Table 13: Request variable usage for erase command

index_u16	data_pu08	bytecount_u16	command_enu
block index inside the FDL pool	unused	unused	FDL_CMD_ERASE_ BLOCK

Table 14: Status of FDL_CMD_ERASE_BLOCK

Status	Class	Status meaning and handling	
FDL_ERR_INITIALIZATION	heavy	meaning	FDL not initialized or not opened
		reason	wrong handling on user side
		remedy	initialize and open FDL before using it
FDL_ERR_STANDBY	heavy	meaning	FDL is in standby and cannot accept new commands
		reason	wrong handling on user side
		remedy	call FDL_WakeUp() before initiating new commands
FDL_ERR_PARAMETER	heavy	meaning	request cannot be accepted
		reason	wrong command code, index outside the used pool or request data structure on odd address
		remedy	correct affected request member and try again
FDL_ERR_ERASE	heavy	meaning	at least one byte within the specified pool area is not “blank”
		reason	internal flash problems
		remedy	do not use this block anymore
FDL_ERR_REJECTED	normal	meaning	request cannot be accepted
		reason	other command is being executed
		remedy	call FDL_Handler() and try again
FDL_ERR_ABORTED	normal	meaning	block oriented command has been aborted
		reason	FDL_Abort() has been called by the user during block command execution
		remedy	restart the erase command via FDL_Execute()
FDL_BUSY	normal	meaning	request is being processed
		reason	request checked and accepted
		remedy	nothing, call FDL_Handler() until status changes
FDL_OK	normal	meaning	request was finished regularly
		reason	no problems during execution
		remedy	nothing

3.5 Basic functional Workflow

To be able to use the FDL (execute pool-related commands) in a proper way the requester has to follow a specific startup and shutdown procedure.

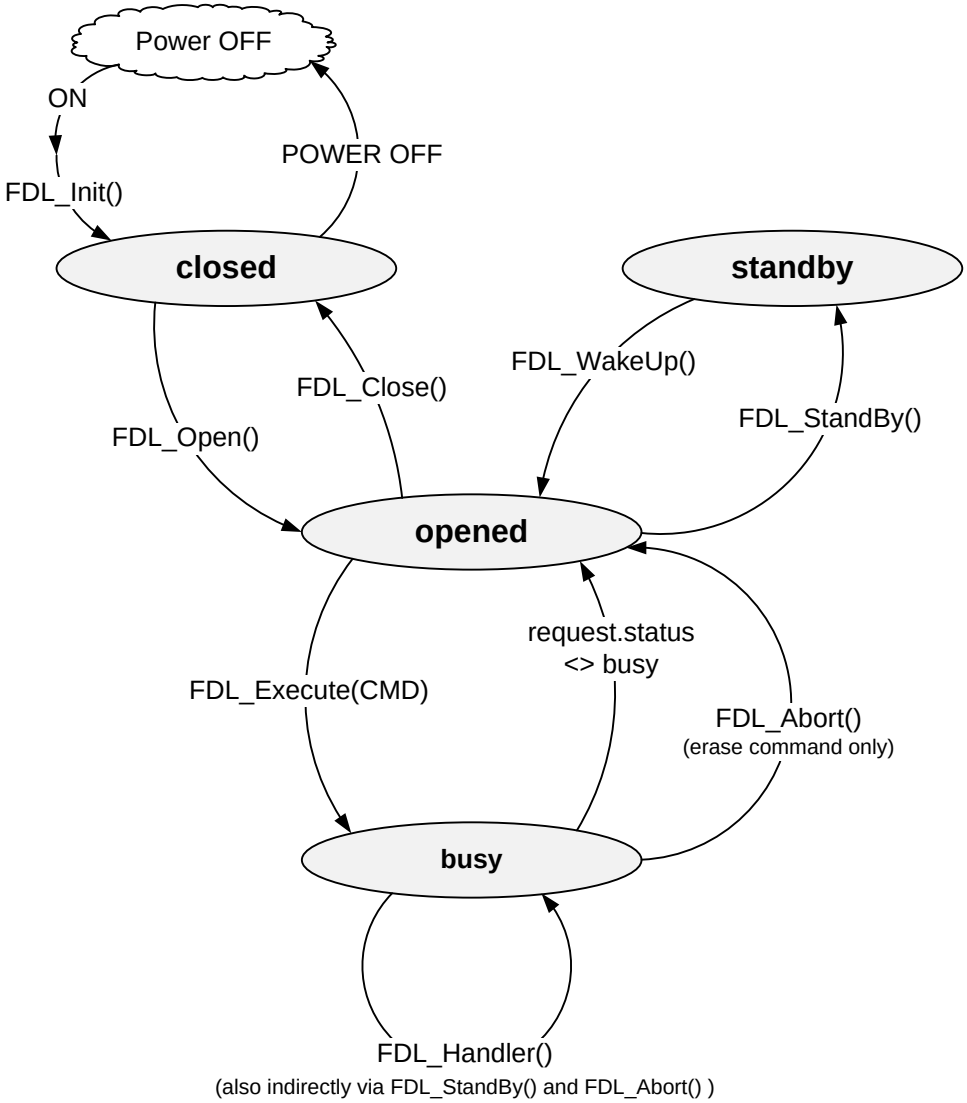

Figure 12: Basic flowchart

Chapter 4 FDL Usage by User Application

4.1 First Steps

It is very important to have theoretic background about the Data Flash and the FDL in order to successfully integrate the library into the user application. Therefore, it is important to read this user manual completely in advance especially chapter “Cautions”.

4.2 Special Considerations

4.2.1 Reset Consistency

During the execution of FDL commands (write and erase), a reset could occur and the data flash content could be damaged. It is designers duty to take care of reset and failure scenarios, e.g. by a proper failure mode and effect analysis. The EEL provided by Renesas Electronics is designed to avoid read of invalid data caused by such reset scenarios. The following chapter describes the applications where the FDL and EEL should be used.

4.2.2 EEL+FDL or FDL only

Depending on the security level of the application, the write frequency of variables and the variable count, it should be considered whether to use the EEL+FDL or the FDL only.

4.2.2.1 FDL only

By using the FDL only the application has to take care about all reset scenarios and writing flow of different variables with different sizes.

Application scenarios

- programming of initial or calibration data
- user specific EEPROM emulation

4.2.2.2 EEL+FDL

The duo of EEL and FDL allows the user to uses the EEL for high write frequency of different variables with different sizes in a secure way and additionally the FDL pool for e.g. application data or application specific EEPROM emulation.

Application scenarios

- programming of initial or calibration data (FDL should be preferred)
- large count of variables with high write frequency (EEL should be preferred)
- reset safe data handling (EEL should be preferred)

Please refer to the R01AN1416EDxxxx manual for detailed EEL description.

4.3 File Structure

The Tiny FDL is delivered as precompiled library for CA78K0R, IAR, GNU and CC-RL environments. The library and its header files are stored in the *lib* subdirectory inside the installation folder. The *smp* directory contains sample setups which are no integral part of the library itself and should be modified according to the project needs. All files are listed in Table 15.

Table 15: File structure of the Tiny FDL

File	Description
<installation folder>	
fdl_info.txt	contains release-specific information about the installed library
<installation folder>/lib	
fdl.h	FDL header file, FDL interface definition
fdl_types.h	FDL header file, FDL types definition
fdl.lib (CA78K0R) fdl.r26 (IAR) fdl.a (GNU) fdl.lib (CC-RL)	precompiled library file
<installation folder>/smp/C	
fdl_descriptor.c	descriptor calculation part
fdl_descriptor.h	pool configuration part
fdl_sample_linker_file.dr (CA78K0R) fdl_sample_linker_file.xcl (IAR) fdl_sample_linker_file.ld (GNU) fdl_sample_linker_file.sub (CC-RL)	linker sample file
<installation folder>/smp/asm	
fdl_descriptor.asm	descriptor calculation part
fdl_descriptor.inc	pool configuration part
fdl_sample_linker_file.dr (CA78K0R) fdl_sample_linker_file.sub (CC-RL)	linker sample file

Note:
Assembler-interface files are available for the CA78K0R and CC-RL Compiler only.

4.4 Configuration

4.4.1 Linker Sections

Following segments are defined by the library and must be configured via the linker description file.

FDL_CODE	Segment for library code. Can be located anywhere in the code flash.
FDL_CNST	Segment for library constants like descriptor. Can be located anywhere in the code flash.
FDL_SDAT	Segment for library data. Must be located inside the SADDR RAM

Note: Please refer to the Chapter 6 and device user's manual for restrictions of RAM and ROM usage.

4.4.2 Descriptor Configuration (Partitioning of the Data Flash)

Before the FDL can be used, the FDL pool and its partitioning have to be configured first. The descriptor is defining the physical/virtual addresses and parameter of the pool which will be automatically calculated by using the FDL_POOL_BLOCKS and EEL_POOL_BLOCKS definition.

Because the physical starting address of the data flash is fixed by the hardware, the user can only determine the total size of the pool expressed in blocks. Also the physical size of the pool is limited by the hardware and must not be defined by the user. Also, the physical size of a flash block is a predefined constant determined by the used hardware.

The first configuration parameter is FDL_POOL_BLOCKS. The minimum value is 0 and means any access to the FDL-pool is closed. The maximum value is the data flash size expressed in blocks in case EEL pool is not used.

The second configuration parameter is EEL_POOL_BLOCKS, the size of the EEL-pool used exclusively for Renesas EEPROM emulation library only. When proprietary EEPROM emulation library is used the EEL-pool shall be set to 0. The maximum size of the EEL-pool is the data flash size build on the device.

Note:

- The virtual address 0 of the FDL pool corresponds with the successor of the last EEL-pool bytes.

4.4.3 Prohibited RAM Area

The Tiny FDL may use a fraction of the user RAM as working area, referred as prohibited RAM area. The size and position of this area is strictly device dependent (many devices do not even have this area) and vary between the different RL78 products. For details, please refer to the document "User's Manual: Hardware" of your RL78 product.

If a prohibited RAM area is specified for the utilized device, it is not allowed to access this area while the Tiny FDL is active. Whenever FDL functions are called, the data in the prohibited area may be rewritten.

4.4.4 Register Bank

The CA78K0R, CC-RL and IAR releases of the FDL use the registers of the currently selected register bank. No implicit register bank switch is performed by the library.

For the GNU release of the FDL, it is mandatory that register bank 0 is active on function entry. No implicit register bank switch is performed by the library. Return values are placed in register bank 1. For details on GNU calling conventions, please refer to the GNU documentation for RL78 devices.

4.4.5 Stack and Data Buffer

The Tiny FDL utilizes the same stack as specified in the user application. It is the developer's duty to reserve enough free stack for the operation of both, user application and FDL.

The data buffer used by the Tiny FDL refers to the RAM area in which data is located that is to be written into the data flash and where data is to be copied to when read is performed. These buffers need to be allocated and managed by the user.

Note: In order to allocate the stack and data buffer to a user-specified address, please utilize the link directives of your framework.

Caution: In contrast to the internal FDL data (FDL_SDAT segment), both stack and data buffer may not be allocated in the short address range from 0xFFE20 to 0xFFEFF—and also not in the prohibited RAM area, if it exists in the target device.

4.4.6 Request Structure

Depending on the user application architecture more than one request variable could be necessary.

e.g.: in case of accessing the EEL from different tasks.

4.5 General Flow

4.5.1 Initialization

The following figure illustrates the initialization flow.

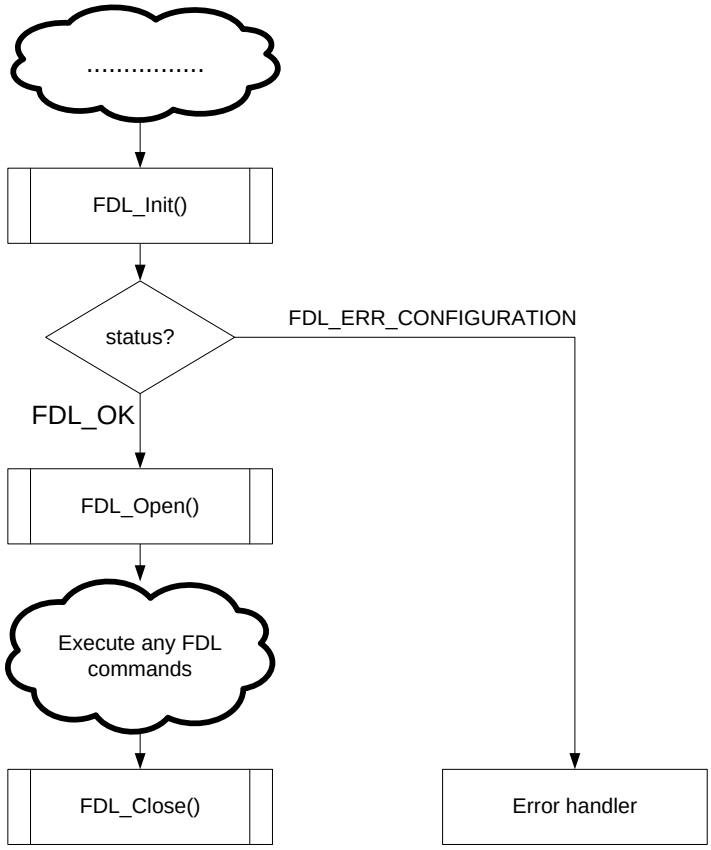

Figure 13: Initialization flow

4.5.2 Read

The following figure illustrates the read command handling.

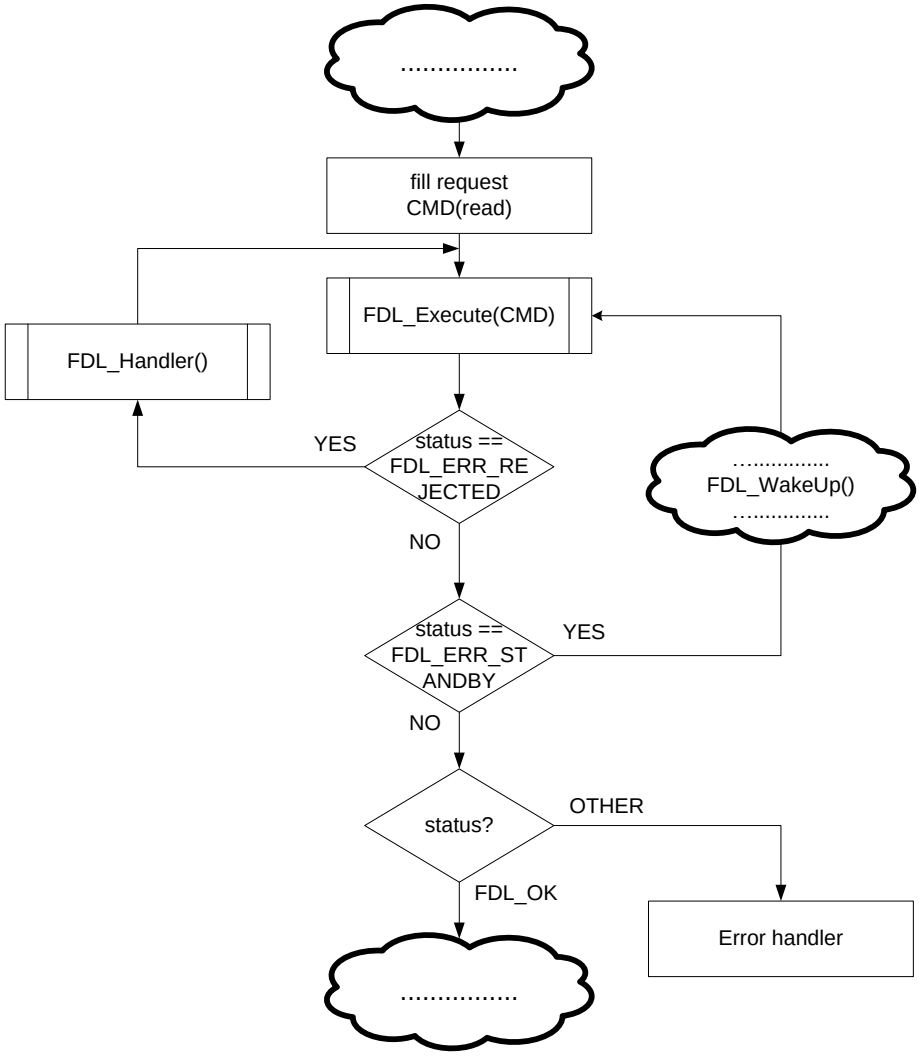

Figure 14: FDL read command execution

4.5.3 Blankcheck/Write/IVerify/Erase

The following figure illustrates the blankcheck/write/internal verify/erase command flow.

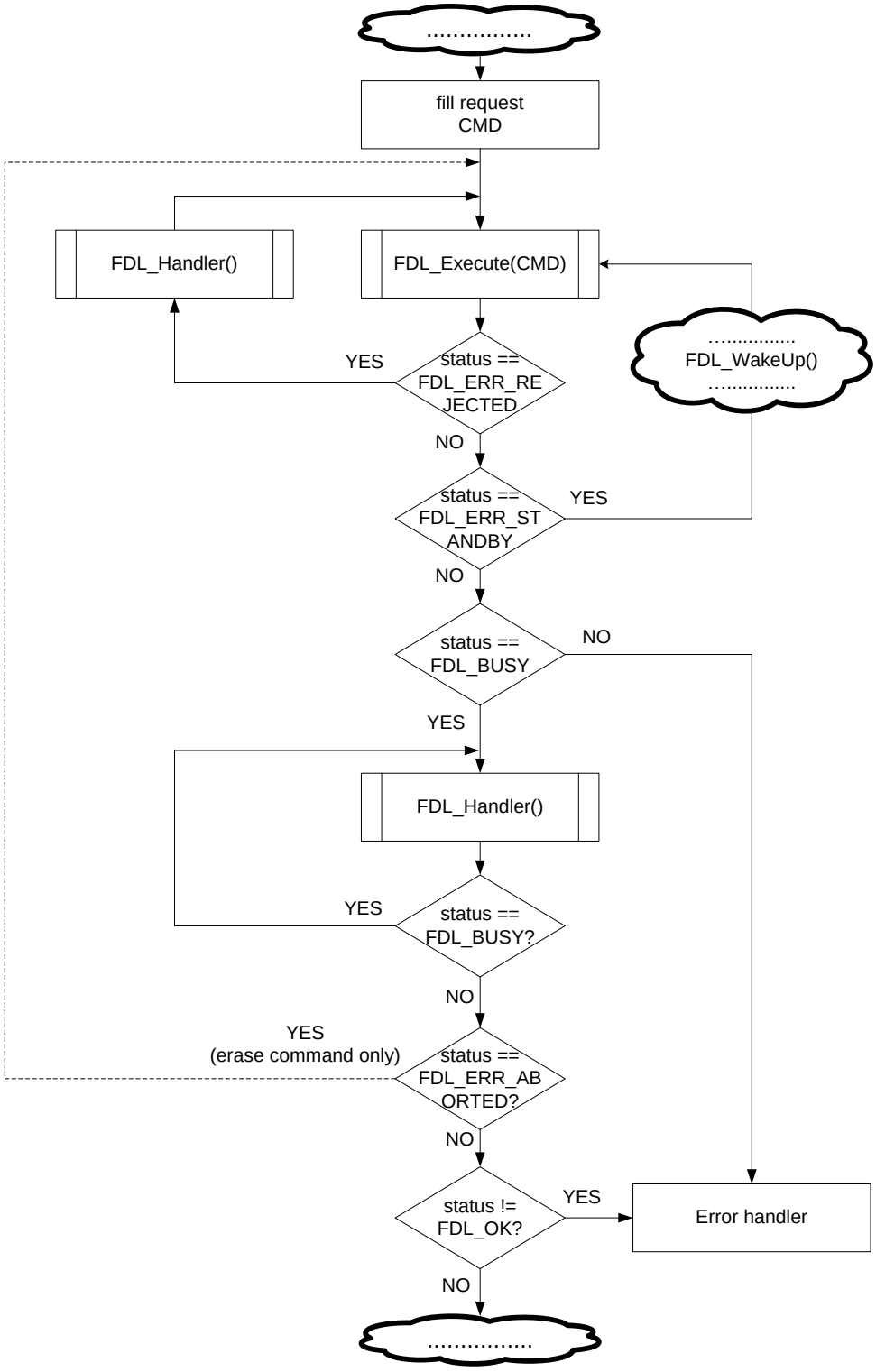

Figure 15: Background operation (Internal-Verify/Blankcheck)

4.6 Example of FDL used in Operating Systems

The background-operation feature and request-response structure of the FDL allow the user to use the FDL in an efficient way in operating systems.

Note: Please read the chapter “Cautions” carefully before using the FDL in such operating systems.

The following figure illustrates a sample operating system where the FDL is used for Data Flash access.

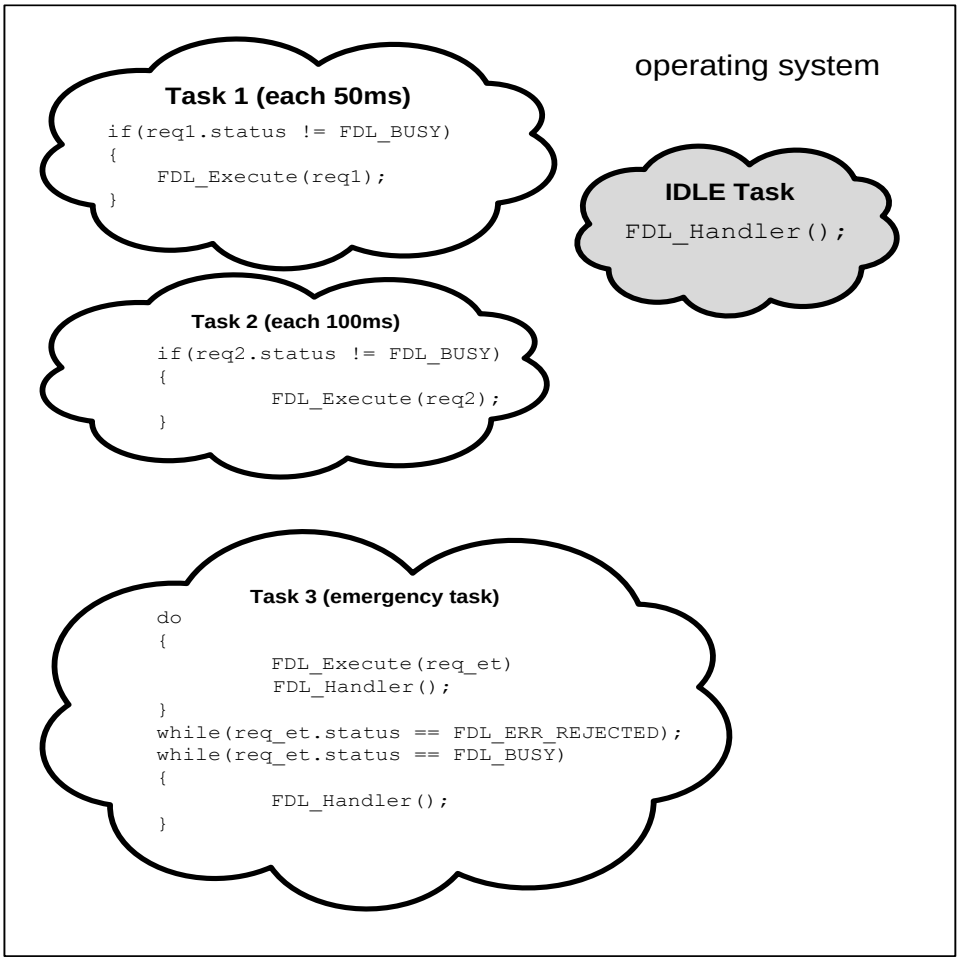

Figure 16: FDL used in an operating system

This operating system example shows three different task types which are described below.

Requesting tasks

Examples of this type of task are Task 1 and 2. Such tasks just start any FDL command via the FDL_Execute function and assume that it will be finished in the background via the IDLE task. Please note that the mandatory error handling is not shown in this example for the sake of readability.

IDLE task

The IDLE task will be used by the application for continuing any running FDL command. That means the FDL_Handler must be called inside of such a task.

Emergency task

The difference between this task type and the requesting type (Task 1 and Task 2) is that this task performs any FDL commands completely without waiting in the background. Such task can be used in

case of voltage drop where important data must be saved before the device is off. Please note that designer could use the abort feature here(for details see Chapter 2.6).

4.7 Example: Simple application

The following sample shows how to use each command in a simple way.

```
extern __far const fdl_descriptor_t   fdl_descriptor_str;
fdl_status_t                          my_fdl_status_enu;
__near fdl_request_t                  request;
fdl_u08                               buffer[5];

/* initialization */
my_fdl_status_enu = FdL_Init(
                    (__far fdl_descriptor_t*)&fdl_descriptor_str );

if(my_fdl_status_enu != FDL_OK) ErrorHandler();
FDL_Open();

/* request structure initialization */
request.index_u16        = 0x0000;
request.data_pu08        = (__near fdl_u08*) 0x0000;
request.bytecount_u16    = 0x0000;
request.command_enu      = (fdl_command_t)0xFF;
request.status_enu       = FDL_ERR_PARAMETER;

/* erase block 0 */
request.index_u16        = 0x0000;
request.command_enu      = FDL_CMD_ERASE_BLOCK;
FDL_Execute(&request);
while(request.status_enu == FDL_BUSY) FDL_Handler();
if(request.status_enu != FDL_OK)      ErrorHandler();

/* write pattern 0x123456789A to idx = 0 */

buffer[0] = 0x12;
buffer[1] = 0x34;
buffer[2] = 0x56;
buffer[3] = 0x78;
buffer[4] = 0x9A;

request.index_u16        = 0x0000;
request.data_pu08        = (__near fdl_u08*)&buffer[0];
request.bytecount_u16    = 0x0005;
request.command_enu      = FDL_CMD_WRITE_BYTES;
FDL_Execute(&request);
while(request.status_enu == FDL_BUSY) FDL_Handler();
if(request.status_enu != FDL_OK)      ErrorHandler();

/* set initial values */
buffer[0] = 0xFF;
buffer[1] = 0xFF;
buffer[2] = 0xFF;
buffer[3] = 0xFF;
buffer[3] = 0xFF;

request.index_u16        = 0x0000;
request.data_pu08        = (__near fdl_u08*)&buffer[0];
request.bytecount_u16    = 0x0005;
request.command_enu      = FDL_CMD_READ_BYTES;
FDL_Execute(&request);
if(request.status_enu != FDL_OK)      ErrorHandler();

FDL_Close();
```

Chapter 5 Characteristics

5.1 Resource Consumption

All values are based on the FDL version V1.01 for CA78K0R/GNU Compiler and FDL version V1.02 for IAR Compiler.

Table 16: Resource consumption

	CA78K0R Compiler	IAR Compiler	GNU Compiler	CC-RL Compiler
Max. code size (code flash)	572 bytes	591 bytes	596 bytes	548 bytes
Constants (code flash)	10 bytes	10 bytes	10 bytes	10 bytes
Internal data (SADDR RAM)	2 bytes	2 bytes	2 bytes	2 bytes
Max. stack (RAM)	56 bytes	56 bytes	60 bytes	48 bytes

5.2 Library Timings

In the following, certain timing characteristics of the Tiny FDL are specified. All timing specifications are based on the following library versions:

- Tiny FDL: V1.00

Please note that there might be deviations from the specified timings in case you are using other library versions than the ones mentioned.

5.2.1 Maximum Function Execution Times

The maximum function execution times are listed in the following tables. These timings can be seen as worst case durations of the specific Tiny FDL function calls and therefore can aid the developer for time critical considerations, e.g. when setting up the watchdog timer. Please note however, that the typical and minimum function execution times can be much shorter.

Table 17: Maximum function execution times (full speed mode)

Function	MAX
FDL_Init	1199/fclk
FDL_Execute (read command)	167/fclk + (17/fclk × BYTE_COUNT)
FDL_Execute (non-read command)	646/fclk
FDL_Handler	284/fclk + 15us
FDL_Open	27/fclk + 14us
FDL_Close (no command running)	30/fclk
FDL_Close**1 (running command in background)	836/fclk + 444us
FDL_StandBy	305/fclk + 15us
FDL_WakeUp	32/fclk + 14us
FDL_Abort	350/fclk + 28us
FDL_GetVersionString	14/fclk

Note **1: It is not recommended to call the FDL_Close function in case of any running command in background.

Table 18: Maximum function execution times (wide voltage mode)

Function	MAX
FDL_Init	1199/fclk
FDL_Execute (read command)	167/fclk + (17/fclk × BYTE_COUNT)
FDL_Execute (non-read command)	646/fclk
FDL_Handler	284/fclk + 15us
FDL_Open	27/fclk + 14us
FDL_Close (no command running)	30/fclk
FDL_Close**1 (running command in background)	791/fclk + 969us
FDL_StandBy	305/fclk + 15us
FDL_WakeUp	32/fclk + 14us
FDL_Abort	350/fclk + 40us
FDL_GetVersionString	14/fclk

Note **1: It is not recommended to call the FDL_Close function in case of any running command in background.

5.2.2 Command Execution Times

The command execution times are listed in the following tables. These timings are divided into the typical timings which will appear during the normal operation and the max timings for worst case considerations.

Table 19: Command execution times (full speed mode)

Command	TYP	MAX
blank-check	1000/fclk + 30us + (5/fclk + 0.26us) × BYTE_COUNT	1200/fclk + 36us + (6/fclk + 0.31us) × BYTE_COUNT
internal verify	715/fclk + 25us + (24/fclk + 3.33us) × BYTE_COUNT	858/fclk + 30us + (29/fclk + 4.00us) × BYTE_COUNT
read	139/fclk + (14/fclk × BYTE_COUNT)	167/fclk + (17/fclk × BYTE_COUNT)
write	580/fclk + 12us + (212/fclk + 39.17us) × BYTE_COUNT	696/fclk + 14us + (714/fclk + 430.00us) × BYTE_COUNT
erase	11344/fclk + 5800us	281674/fclk + 264790us

Table 20: Command execution times (wide voltage mode)

Command	TYP	MAX
blankcheck	996/fclk + 63us + (5/fclk + 0.90us) × BYTE_COUNT	1196/fclk + 75us + (6/fclk + 1.05us) × BYTE_COUNT
internal verify	715/fclk + 49us + (15/fclk + 24.17us) × BYTE_COUNT	858/fclk + 58us + (18/fclk + 29.00us) × BYTE_COUNT
read	139/fclk + (14/fclk × BYTE_COUNT)	167/fclk + (17/fclk × BYTE_COUNT)
write	580/fclk + 12us + (209/fclk + 82.50us) × BYTE_COUNT	696/fclk + 14us + (670/fclk + 954.00us) × BYTE_COUNT
erase	10019/fclk + 7195us	249113/fclk + 299307us

Chapter 6 Cautions

- Library code and constants must be located completely in the same 64k flash page.
  - For CA78K0R compiler, the library takes care in the code to define these sections with UNIT64KP relocation attribute.
  - For CC-RL compiler, the library takes care in the code to define these sections with TEXTF_UNIT64KP relocation attribute.
  - For IAR compiler, the user has to ensure that the linker file specifies the Flash page size equal to 64K when defining FDL_CODE and FDL_CNST sections.
  - For GNU compiler, the user shall take care that FDL_CODE and FDL_CNST sections are not mapped across any boundary of 64KB Flash page.
- The library initialization by FDL_Init must be performed before the execution of FDL_Open, FDL_Close, FDL_Handler, FDL_Execute, FDL_Abort, FDL_StandBy and FDL_WakeUp.
- It is not allowed to read the data flash directly (meaning without FDL) during a command execution of the FDL.
- Each request variable must be located at an even address.
- Before executing any command, all members of the request variable must be initialized. If there are any unused members in the request variable, please set arbitrary values to these members.
- All functions are not re-entrant. That means it is not allowed to call FDL functions inside the ISRs while any FDL function is already running.
- Task switches, context changes and synchronization between FDL functions:
  - All FDL functions depend on FDL global available information and are able to modify this information. In order to avoid synchronization problems, it is necessary that at any time only one FDL function is executed. So, it is not allowed to start an FDL function, then switch to another task context and execute another FDL function while the last one is not yet finished.
  - Example for a not allowed sequence:
    - Task 1: Start an FDL operation with FDL_Execute.
    - Interrupt the execution and switch to task 2, executing FDL_Handler.
    - Return to task 1 and finish the FDL_Execute function.
- After the execution of FDL_Close, all requested/running commands will be aborted and cannot be resumed. The designer has to take care that all running commands are finished before calling FDL_Close.
- It is not possible to modify the Data Flash via FDL in parallel to a modification of the Code Flash via FSL.
- An abortion of the byte commands read, write, verify, and blankcheck is not possible.
- Internal high-speed on-chip oscillator (HOCO) must be started before using the FDL.
- It is not allowed to locate any arguments and stack memory to address of 0xFFE20 and above.
- In case the application requires a frequency of less than 4MHz, the following frequencies are allowed: 1MHz, 2MHz, 3MHz. It is not allowed to use the frequency of e.g. 1.5MHz. The library configuration parameter FDL_SYSTEM_FREQUENCY in "FDL_descriptor.h" shall be adapted according to the above definition (e.g. 1000000, 2000000, 3000000).
- In case the Data Transfer Controller(DTC) is used in parallel to the FDL, do not locate RAM area for DTC to address 0xFFE20 and above
- Please check the device restrictions described in the device user’s manual in case of accessing the data flash via the FDL
- Execution of byte commands (blankcheck, internal verify, write and read) across block boundaries is not allowed

Revision History

Chapter	Page	Description
		Initial revision
		Revision V1.10:
3.2	23	Add a more detailed description of types and their meaning
3.3	27	Adding description of GNU API
4.3	51	Extend file structure for GNU
4.4.4	52	Add register bank selection chapter
5.1	59	Updated resource consumption information
6	62	Explain how library can be mapped in the same 64K flash page for each compiler
		Revision V1.20:
3.3	28	Adding description of CC-RL API
4.3	54	Extended file structure for CC-RL
all	all	Renesas (REN) Compiler renamed to CA78K0R
3.4.x/3.4 5.1	all 62	Byte count range specification added Resource consumption added for CC-RL

Data Flash Access Library

Renesas Electronics Corporation

R01US0061ED0120